Fundamentals of Package Development

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

- WiFi:
  - Network: Posit Conf 2023
  - Password: conf2023
- Bathrooms through the kitchen
- There are **gender-neutral bathrooms** located 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 colours 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
- 844-448-1212

Welcome!

- Introductions
  - Instructor:
    - Andy Teucher
    - GitHub: ateucher
    - Mastodon: @andyteucher@fosstodon.org
  - TAs:
    - Nic Crane
    - Simon Couch
  - Yourselves

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.

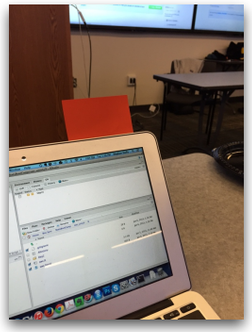
Sticky Notes

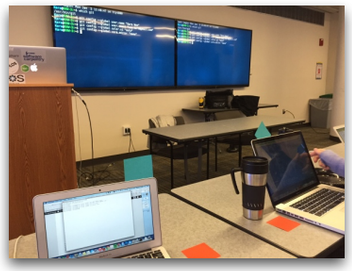
PURPLE

I'm stuck, please send help

GREEN

I'm good, let's move on





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!

Resources

- Workshop website: pos.it/pkg-dev-conf23
- Cheatsheet in your handout

Schedule and Learning Objectives

START: 9:00

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

BREAK: 10:30 - 11:00

- Package Creation and Metadata
- Documentation

LUNCH: 12:30 - 1:30

- Testing
- Package Dependencies

BREAK: 3:00 - 3:30

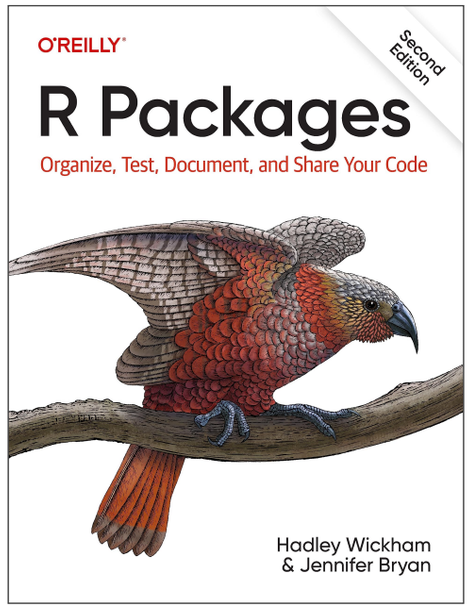
- Continuous Integration
- Package Website & Vignettes
- Package Distribution (CRAN)

END: 5:00

R Packages (2e)

Hadley Wickham
Jenny Bryan

https://r-pkgs.org



The core ideas in this course are taken from the R Packages by Hadley Wickham and Jenny Bryan, now in its second edition.
We basically follow the format of the "Whole Game" chapter, stepping through the key steps in making a package, with detours along the way to explore the main concepts in a bit more detail.

Packages in a nutshell 

Why make a package?

Why make a package?

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 channel

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

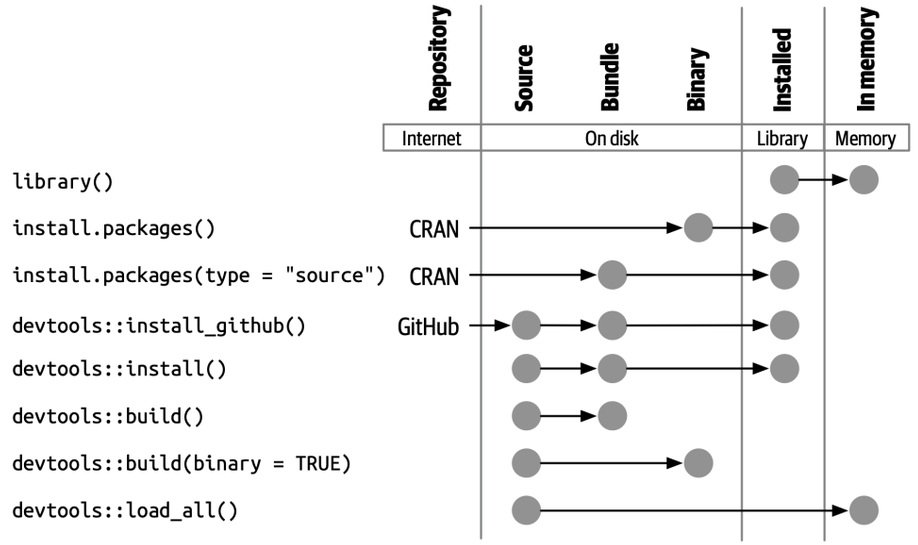
- How many libraries do you have?
- What are they? (Put them in the Discord)

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



Wickham, Hadley, and Jennifer Bryan. 2023. R Packages. 2nd ed.

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)
- Focus on workflows

We won't:

- Talk (much) about function writing and design
- 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.master
- A package to explore our local R package libraries

** Put link in Discord**
This is the package in “Source” form:
- Package code in R/
- Function documentation files in man/
- Package vignettes in vignettes/
- DESCRIPTION
- NAMESPACE

libminer

Sneak peak of our end goal on GitHub

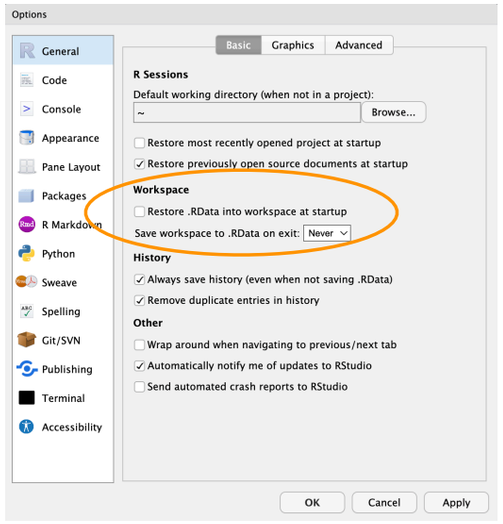
- https://github.com/ateucher/libminer.master
- A package to explore our local R package libraries

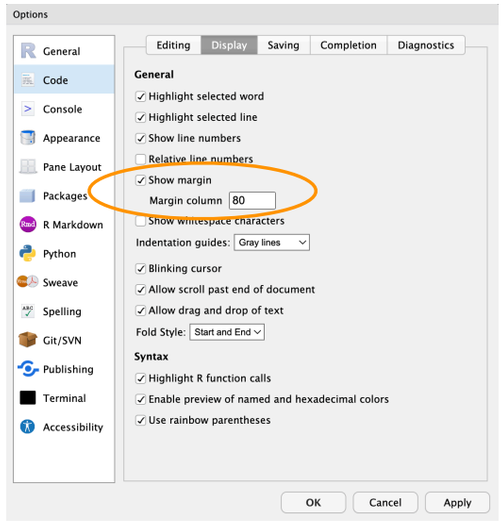
Get Ready

Get Ready

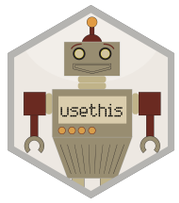
Configure RStudio

Tools > Global Options





Tools



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

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

usethis provides main “workflow” tools

Devtools:

- combines tools for creating, building, loading, installing, checking etc. packages
- Exposes usethis

Create a package

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

Don’t go to script here

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>'
```

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 them?
#>
#> 1: Absolutely not
#> 2: Not now
#> 3: Yeah
```

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
#>
#> 1: Ab
#> 2: No
#> 3: Ye
```

Your Turn

devtools::use_devtools()

Automatically load devtools when R starts

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

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

⌨ Ctrl+Shift+F10 (Windows & Linux)
⌨ Cmd+Shift+0 (macOS)

Opens .Rprofile - file containing R code that is run when R is opened. Useful for setting options

Very rarely load libraries, but devtools and testthat are different as their functions rarely appear in scripts or used in other functions

`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+0 (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 definition!) in this file

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)

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

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)

Test your function in the new package

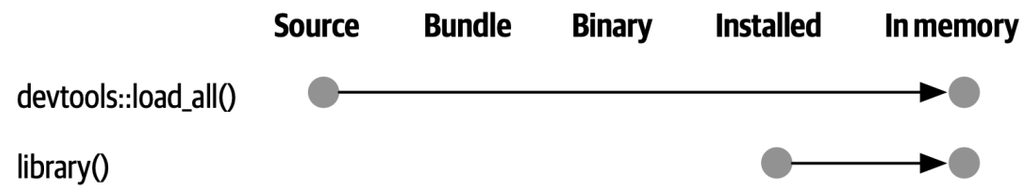
But how?

- ~~`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()



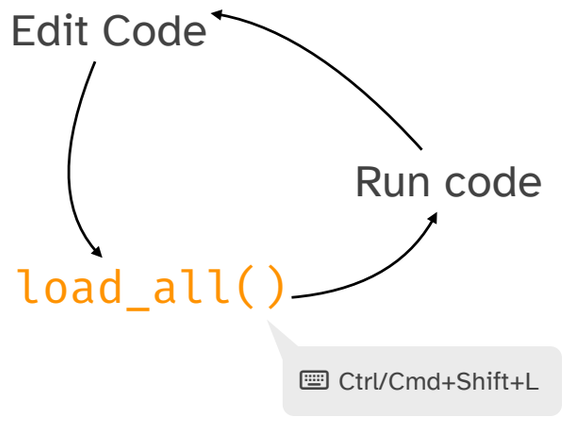
- Simulates building, installing, and attaching your package
- Makes all of the functions from your package immediately available to use
- Allows fast iteration of editing and test-driving your functions
- Good reflection of how users will interact with your package*

Wickham, Hadley, and Jennifer Bryan. 2023. R Packages. 2nd ed.

*With load_all(), unexported functions are also made available, which they are not via install and attach

Try it out, and commit your changes 

Workflow



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*

**If it hurts, do it more often, by Martin Fowler:*
https://martinfowler.com/bliki/FrequencyReducesDifficulty.html

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

https://ma

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.
  - **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'
```

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 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
```

DESCRIPTION is defining feature of a package - every package must have one
Contains important metadata about your package
Specific formatting (DCF: Debian Control Format), machine readable

Title must be one line, title case, no full stop
Description more detailed than the title. must be full sentences, end with a full stop.
Cannot start with “A package for...”

Author field unique because it contains executable R code (person() function)

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 pack
License: `use_mit_license(
   friends to pick a licen
Encoding: UTF-8
Roxygen: list(markdown = T
RoxygenNote: 7.2.3
```

Your Turn

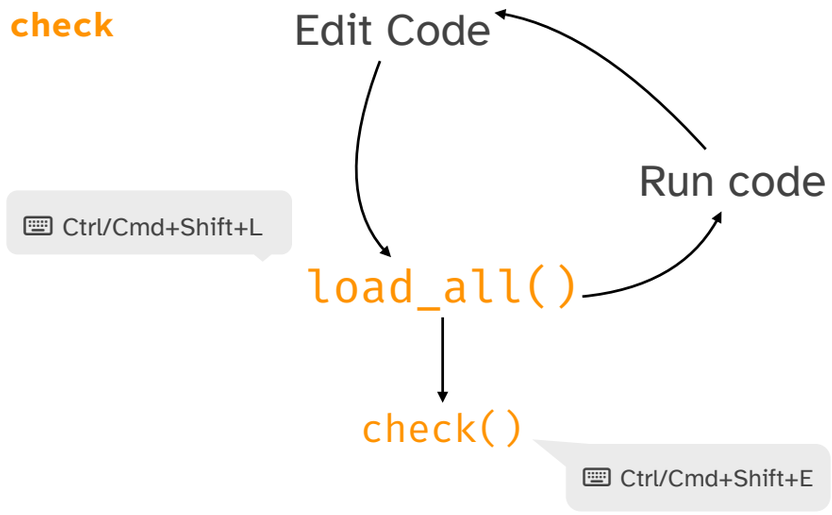
The `DESCRIPTION` file

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



check() again

```
check()


#> ── Documenting ──────────────────────────────────────────────────────────────
…
#> ── Building ─────────────────────────────────────────────────────────────────
…
#> ── Checking ─────────────────────────────────────────────────────────────────
…
#> ── R CMD check results ──────────────────────────────────────────────────────
#> Duration: 3.1s
#>
#> 0 errors ✔ | 0 warnings ✔ | 0 notes ✔
```

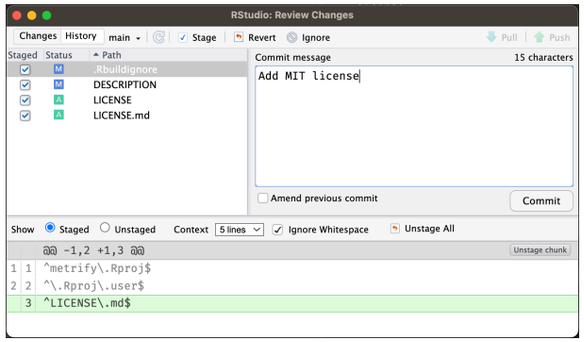
Go to script - check()


TED
LASSO


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 repository on GitHub

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

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

Opens .Rprofile - file containing R code that is run when R is opened. Useful for setting options

Remove the ‘comment’ line if you don’t have an ORCID.
*persistent digital identifier (an ORCID iD) for academic and professional work. You can connect your iD with your professional information — affiliations, grants, publications, peer review, and more.

** Put this template code in Discord

While you're in there...

Set some other helpful defaults

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

Take this opportunity to sneak in some other best practices

Documentation

Documentation

Documentation

Documentation

Function documentation

```
> ?use_git

use_git                package:usethis                R Documentation

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)
```

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}{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()
}
}
\seealso{
Other git helpers:
\code{\link{use_git_config}()},
\code{\link{use_git_hook}()},
\code{\link{use_git_ignore}()}
}
\concept{git helpers}
```

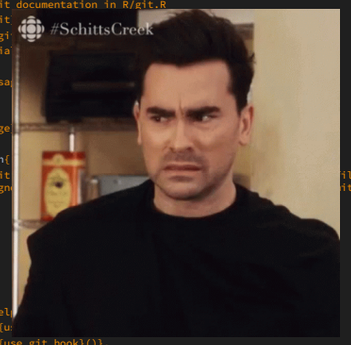
Function documentation

```
> ?use_git

use_git                package:usethis                R Documentation

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)
```

Documentation

`man/*.Rd`

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/git.R
\name{use_git
\alias{use_gi
\title{Initia
\usage{
use_git(messa
}
\arguments{
\item{message
}
\description{
\code{use_git
\code{.gitign
}
\examples{
\dontrun{
use_git()
}
}
\seealso{
Other git hel
\code{\link{u
\code{\link{use_git_hook}()},
\code{\link{use_git_ignore}()}
}
\concept{git helpers}
```



Function documentation

```
> ?use_git

use_git                package:usethis                R Documentation

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)
```

roxygen2

- RStudio: *Code* > *Insert Roxygen Skeleton*

Cmd/Ctrl+Alt+Shift+R

- 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!

```
#' 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

```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 — ⌨ Ctrl+. (Start typing function name...)
- Cursor in function definition
- Insert roxygen skeleton — ⌨ Cmd/Ctrl+Alt+Shift+R
- Complete the roxygen fields
- `document()` — ⌨ Cmd/Ctrl+Shift+D
- `?myfunction`
- 🎉

Create roxygen comments

- Go to function definition — ⌨ Ctrl+. (Start typing function name...)
- Cursor in function definition
- Insert roxygen skeleton — ⌨ Cmd/Ctrl+Alt+Shift+R
- Complete the roxygen fields
- `document()` — ⌨ Cmd/Ctrl+Shift+D
- `?myfunction`
- 🎉

Your Turn

check()
Commit your changes
Push to Github

NAMESPACE

An introduction

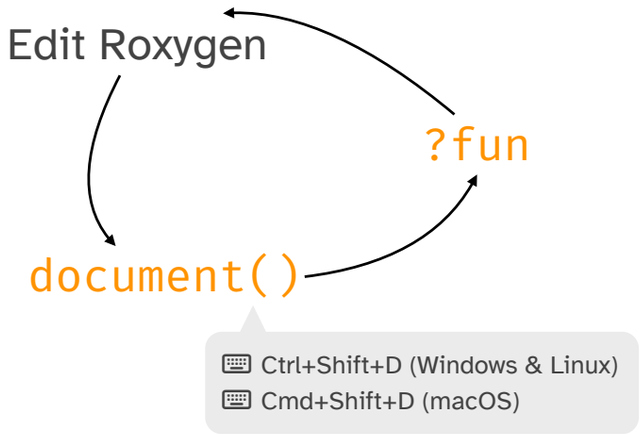
- Lists R objects that are:
  - **Exported** from your package to be used by package users
    - `export(), S3method(), …`
  - **Imported** from another package to be used internally by your package
    - `import(), importFrom(), …`
- `document()` updates the `NAMESPACE` file with directives from Roxygen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

export tag in R code -> export(lib_summary) in NAMESPACE

NAMESPACE file is consulted to determine imports and exports, Roxygen comments are used to populate the NAMESPACE when you call document().

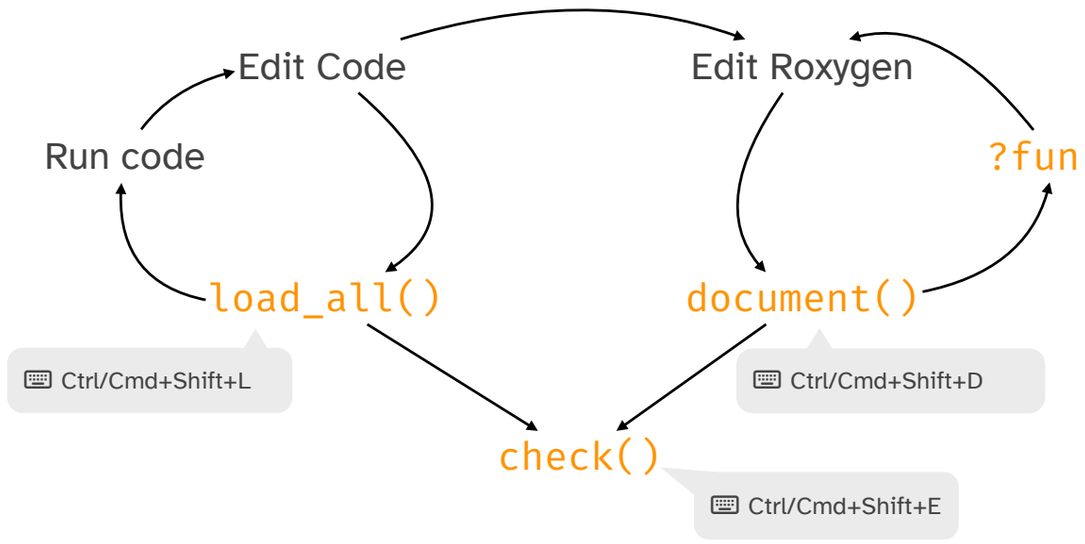
If using roxygen2, DON’T EDIT NAMESPACE file. In fact, most of the time you don’t even need to worry about it because roxygen2 takes care of it for you.

Documentation workflow



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

- Adds a dummy .R file that will prompt roxygen to generate basic package-level documentation.
- Once you call devtools::document(), roxygen will flesh out the .Rd file using data from the DESCRIPTION.
- Ensures you don't need to repeat the same information in multiple places.

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+0 (macOS)
- Attach package with `library()` like any other package

`install()`

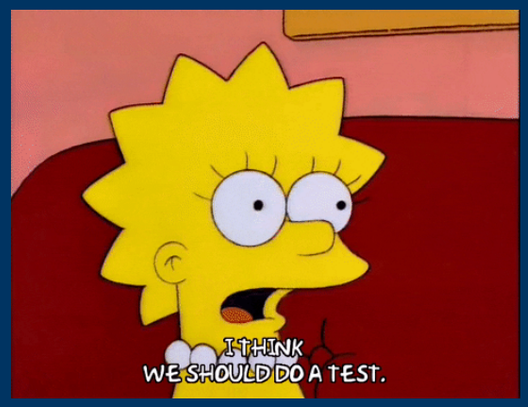
Install package to your library

- `R CMD INSTALL` ⌨ Ctrl+Shift+B (Windows & Linux) ⌨ Cmd+Shift+B (macOS)
- Restart R ⌨ Ctrl+Shift+F10 (Windows & Linux) ⌨ Cmd+Shift+0 (macOS)
- Attach package with `library()` like any other packa

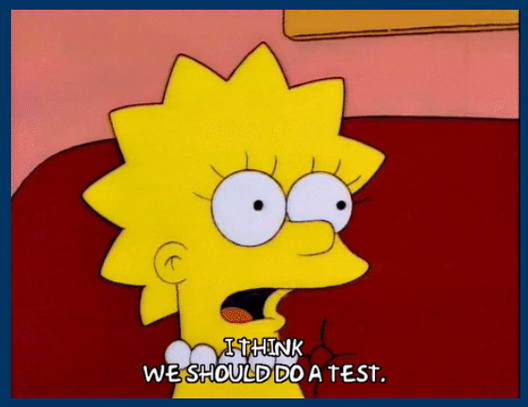
Your Turn

Commit your changes 
Push to Github 

Testing

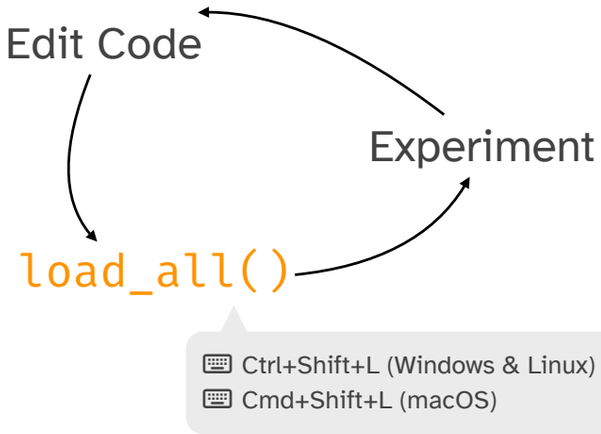


Testing



Testing

Current workflow



The problem with this approach is that when you come back to this code in 3 months time to add a new feature, you've probably forgotten some of the informal tests you ran the first time around. This makes it very easy to break code that used to work.

Automated Testing

Benefits

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

Testing confirms that your code does what you intended - functions return what you expect them to, fail in expected ways, and deal with edge cases that users might throw at them.

Bugs:
- Informal testing, it’s tempting to just explore typical usage, similar to writing examples.
- Formal tests, anticipate how unexpected usage could break your code
- Write new tests when you add a new feature, help prevent bugs from being created in the first place, because you will proactively address pesky edge cases.

Code Structure:
- When you write code with testing in mind, will quickly learn that poorly designed code is harder to test - will help force you into writing writing small, modular functions that are easier to test.

Call to action:
- Write a failing test for new bugs first

Robust code:
- Allows confident refactoring, even major, comfortable in the knowledge that you won’t break anything inadvertently.

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

- Go through each line of output.

An edition is a bundle of behaviours that you have to explicitly choose to use, allowing testthat devs to make otherwise backward incompatible changes.

- Don't edit `tests/testhat.R`

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

Test files are R files that live in `tests/testthat/` and must start with `test-`

recommend that the organization of test files match the organization of R/ files

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

A test failure report includes description of test_that - make description informative.

Expectation: Does it have the right value and right class? Does it produce an error when it should?

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)

`test()`

- Runs all tests in your test suite

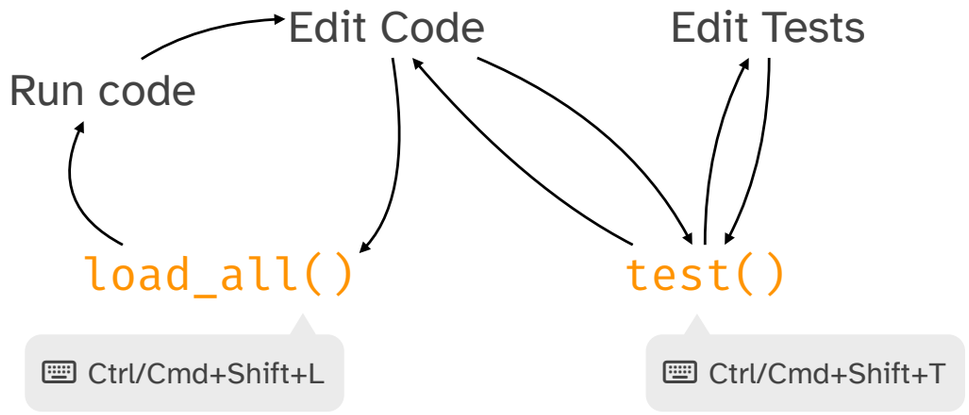
```
test()

#> 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

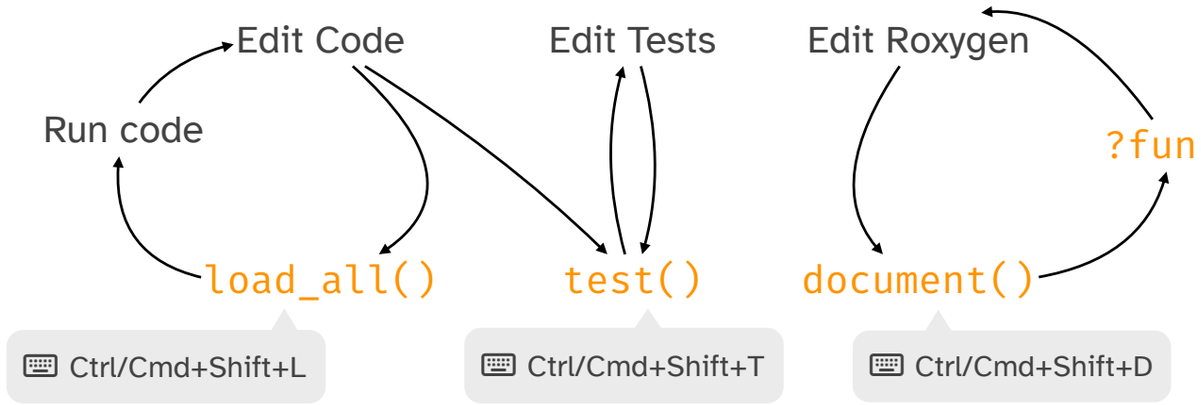
Updated workflow

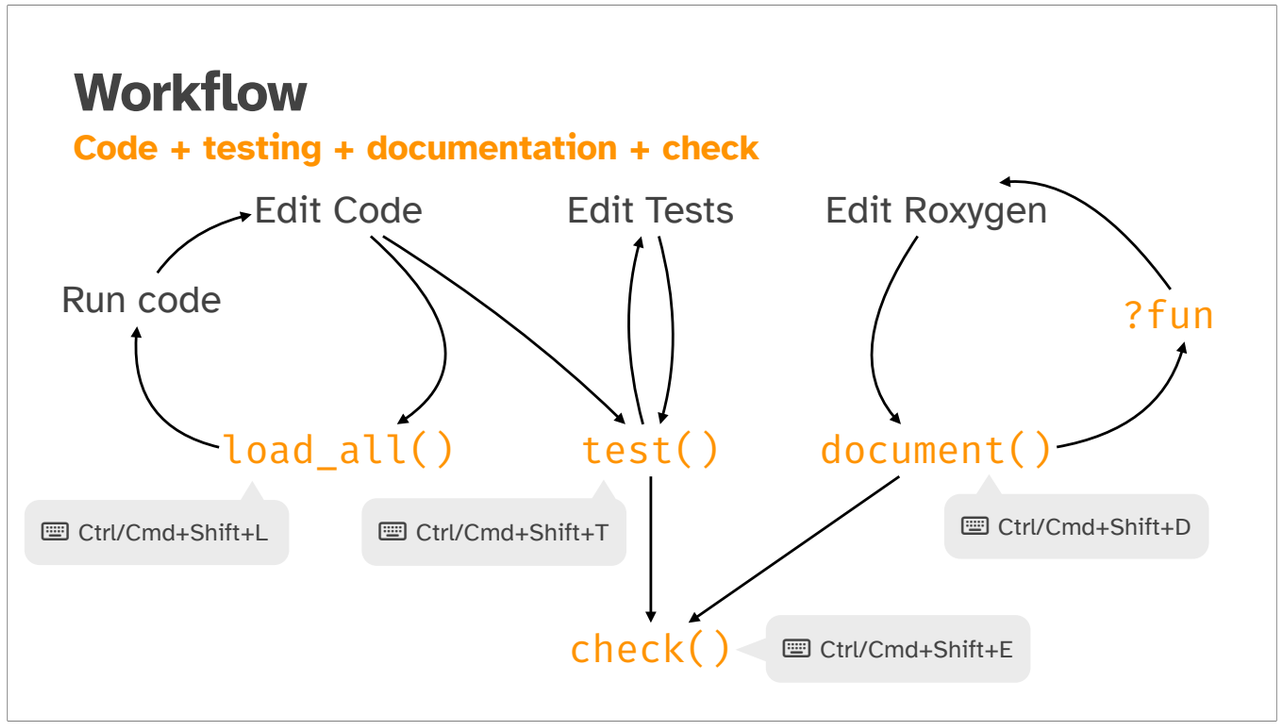
Code + testing



Workflow

Code + testing + documentation





** check() also runs all of your tests **

** To facilitate this workflow, tests should be small and run quickly, or else you won’t run them! **

check()
Commit your changes
Push to Github

Dependencies

Dependencies

Use functions from another package inside your package

```
library("fs")

# use functions from the fs package…
```

Use functions from another package inside your package

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()`
```

Highlight output bullets!

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

By default, use_package() puts your dependency in the Imports field. Most common and recommended, but want to touch on the others.

Depends is aggressive / user-hostile if overused.

See devtools GitHub repo

From here we will assume we're listing our dependencies in Imports as it's the most common.

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

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
```

The use of the word Imports is tricky because it's used in both DESCRIPTION and NAMESPACE files, but have different effects.

*Note that putting deep in Imports in DESCRIPTION does load the package so that pkg::fun() can work, but this nuance is not visible to users so not really worth talking about at this point.

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

Most common and recommended default approach - WHY?

- clear code: immediately clear where functions are coming from
- Reduces chances of name conflicts causing unexpected behaviour

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()`

Useful when:

- Using one or a few functions from a package, but use them a lot - may make your code more readable
- and when low chance of namespace conflicts (i.e., pretty unique names)

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()

Rare

Useful when:

- Using a lot of functions from a package
- Example is rlang in a lot of tidyverse packages

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)
   ```

You always must include it in the DESCRIPTION file!

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

*Check() before running document()

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

Let’s add one more

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

- See:
  - `DESCRIPTION`
  - `R/mypackage-packge.R` (remember `use_package_doc()`?)
  - `NAMESPACE`
- Write/edit function using dependency: `fn()`
- *`test()`

* Note the importance of the test here - ensures that the function still behaves as we expect, even though we’ve changed the internals.

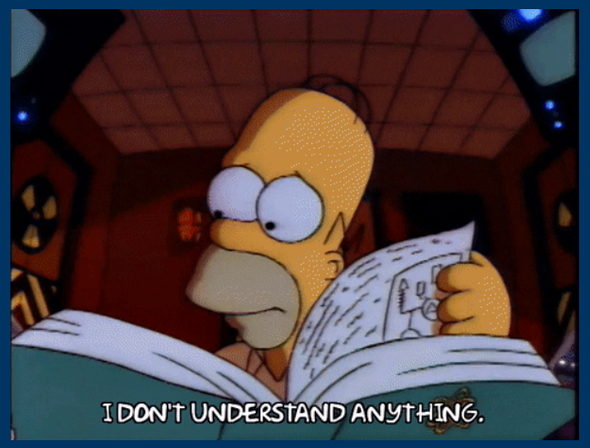
Let's add one more

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

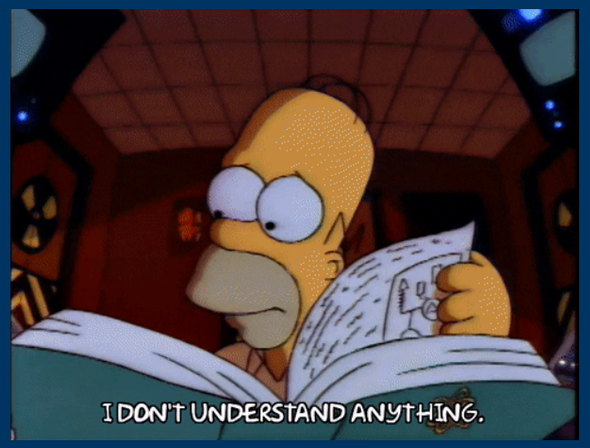
- See:
  - `DESCRIPTION`
  - `R/mypackage-packge.R` (remember `use_package_doc()`?)
  - `NAMESPACE`
- Write/edit function using dependency: `fn()`
- `*test()`

check()
Commit your changes
Push to Github

README



README



`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'
```

Important source of information to people visiting the package's source code home on GitHub

`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

`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

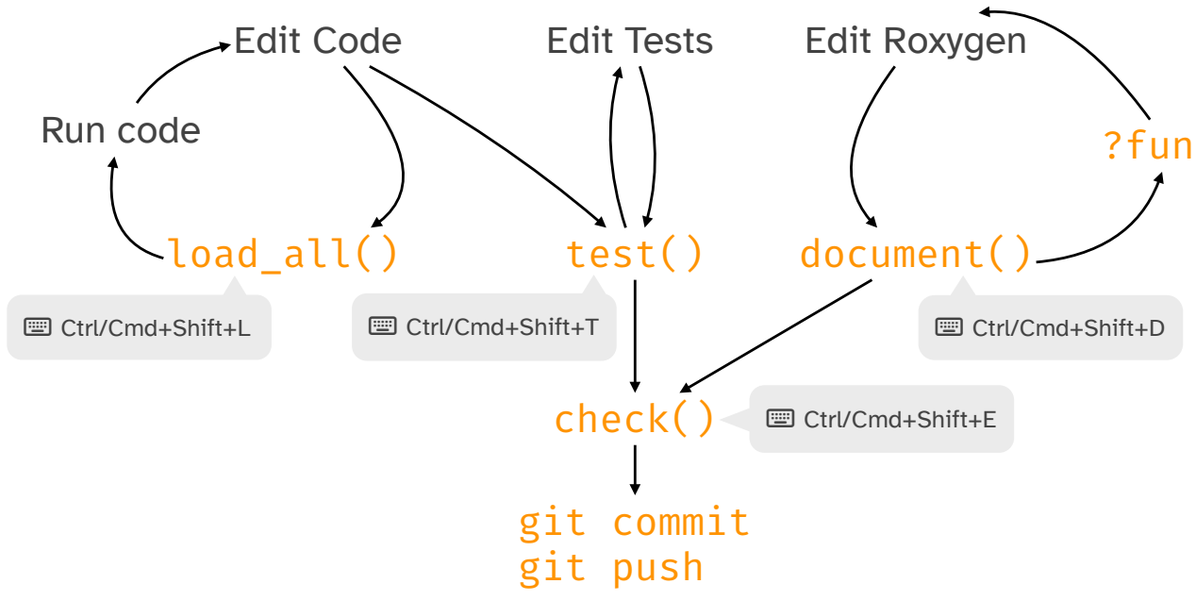
Final `check()` and `install()`

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()`

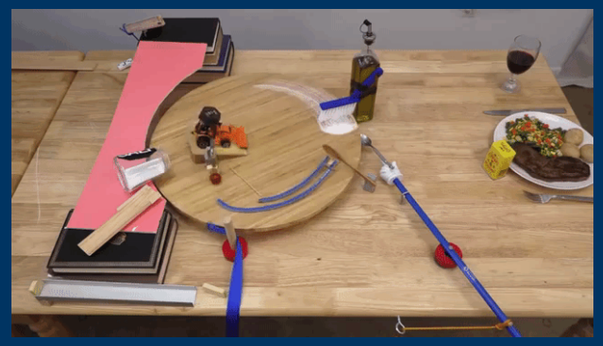
Run periodically

- `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

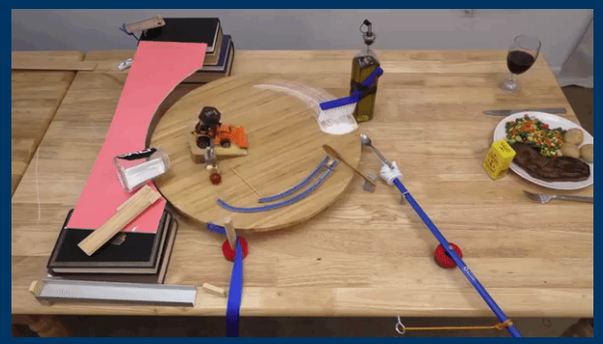
Continuous Integration



Software development practice where you regularly merge your changes into central repository, where some automated processes and tests are run.

In practice, we’re going to use GitHub Actions, a free workflow automation service included with your GitHub account

Continuous Integration



`use_github_action()`

github.com/r-lib/actions

- `"check-standard":` Runs `R CMD check` when you push:
  - On different platforms (macOS, Windows, Linux)
  - On current R, R-devel, and R-oldrel
- `"test-coverage":` Compute test coverage and report at `codecov.io`
- `"pr-commands":` Enables automatic documentation and styling of code via special PR comments
- And more...*

*https://github.com/r-lib/actions/tree/v2-branch/examples

use_github_action()

github.com/r-lib/actions

- `"check-standard"`: Runs `R CMD check` when you push:
  - On different platforms (macOS, Windows, Linux)
  - On current R, R-devel, and R-oldrel
- `"test-coverage"`: Compute test coverage and report at `codecov.io`
- `"pr-commands"`: Enables automatic documentatio
  special PR comments
- And more...*

*https://github.com/r-lib/

Your Turn

Website

Website

`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

1. Initializes an empty, “orphan” branch in your GitHub repo, named gh-pages:
 - only lives on GitHub
 - only tracks files that make up your package’s website (the files that you see locally below docs/).

Turns on GitHub Pages for your repo and tells it to serve a website from the files found in the gh-pages branch.

Copies the configuration file for a GHA workflow that does pkgdown “build and deploy”. The file shows up in your package as .github/workflows/pkgdown.yaml. If necessary, some related additions are made to .gitignore and .Rbuildignore.

Adds the URL for your site as the homepage for your GitHub repo.

Adds the URL for your site to DESCRIPTION and _pkgdown.yml.

`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

Your Turn

Review: functions

Run once

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

Run periodically

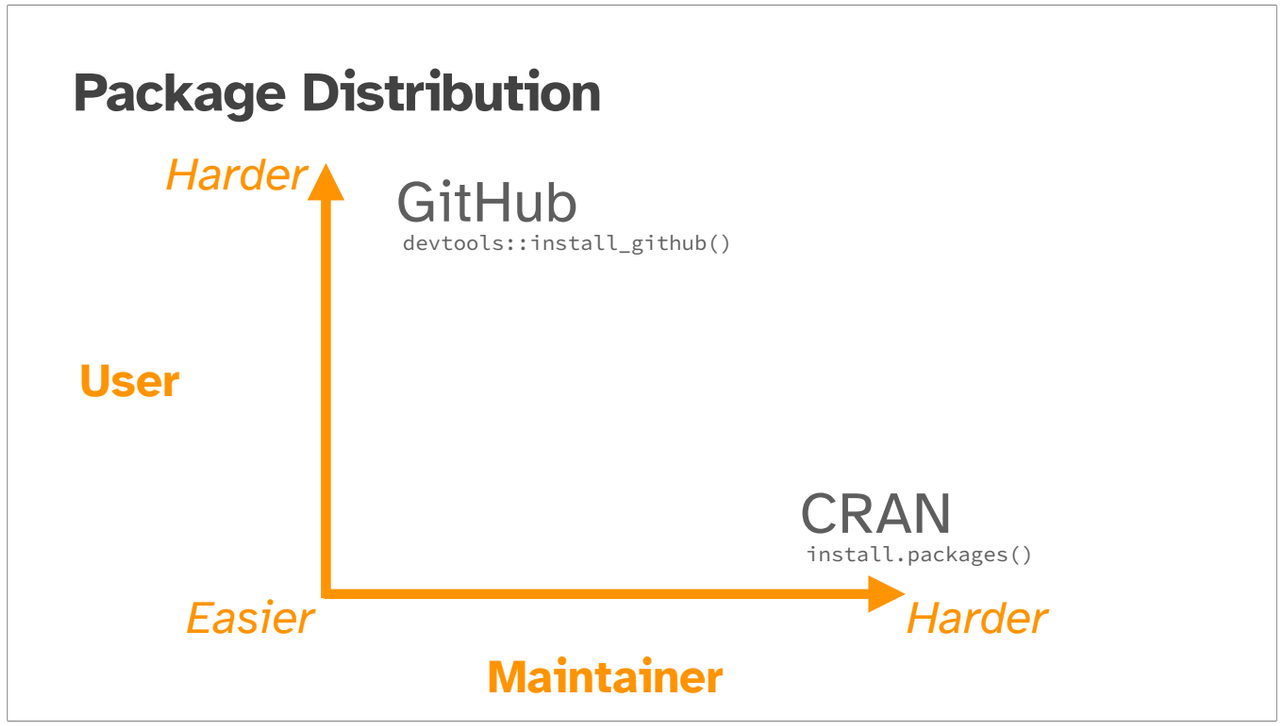
- `use_r()`
- `use_test()`
- `use_package()`
- `rename_files()`
- `use_github_action()`
- `use_vignette()`

Run frequently

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

Sharing your Package

Sharing your Package



Installing from GitHub comes free with the development practices we've discussed

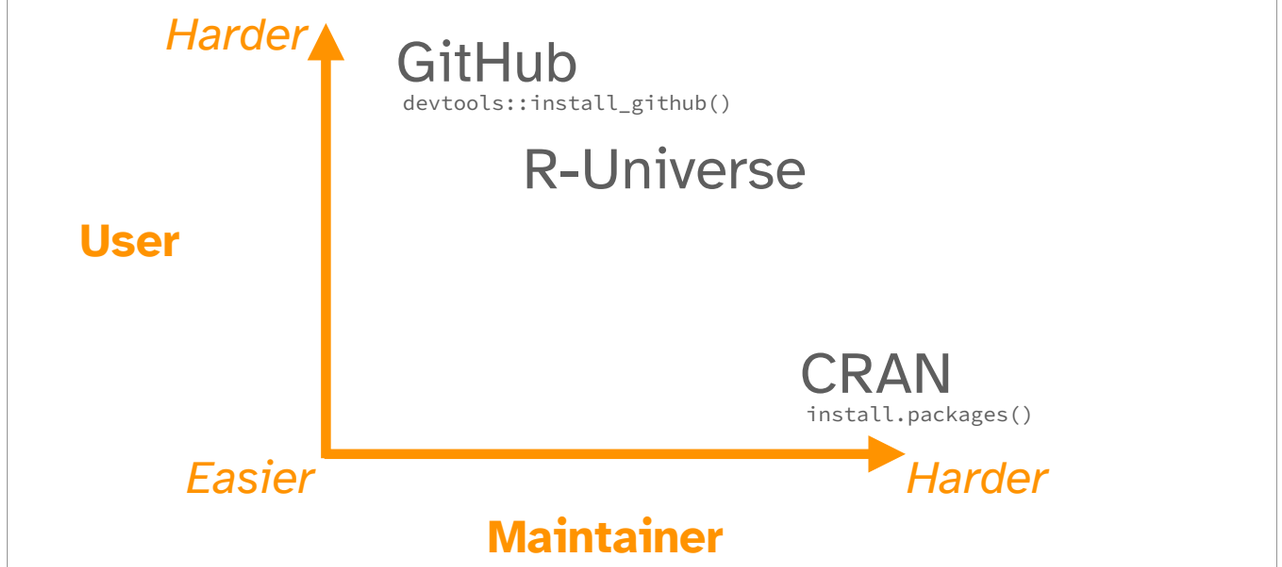
- Great if you have a niche user base who are a bit more savvy
- Need careful development practices to do releases
- How do users find out about updates (no check for package updates in RStudio or update.packages)

Not going to talk about R-Universe today

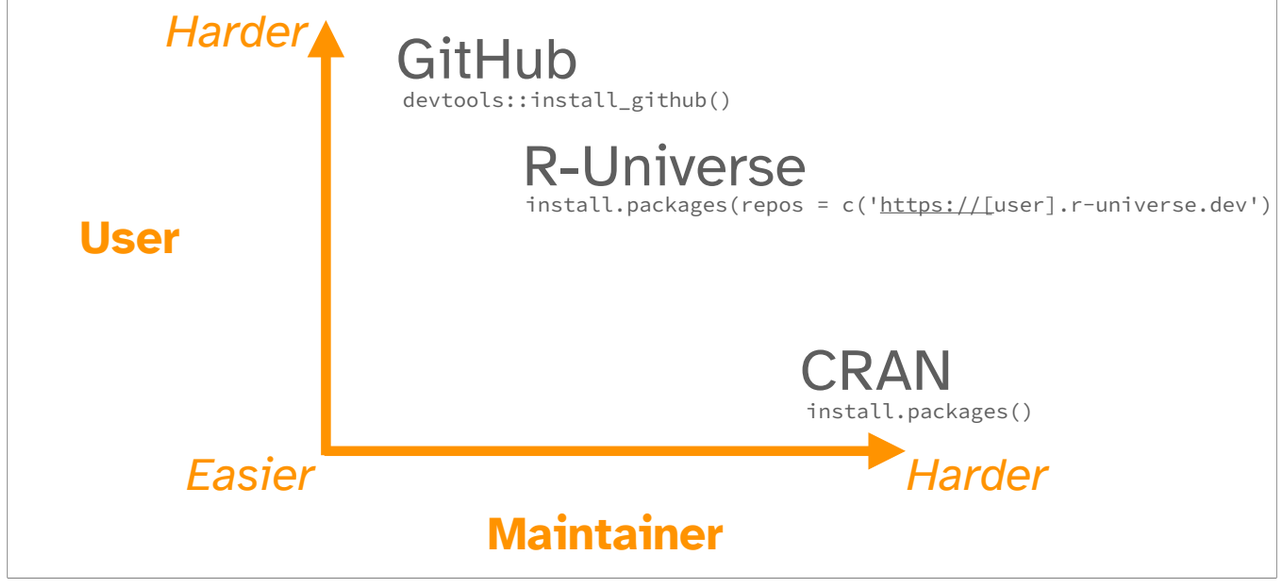
CRAN

- where users expect to get their packages
- Build binaries for easy install
- Updates easier
- Distinct releases

Package Distribution



Package Distribution



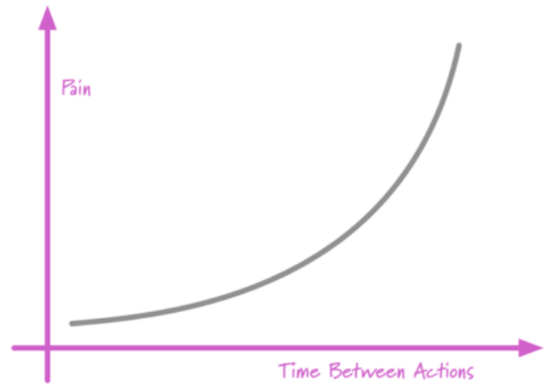
Releasing your package to CRAN

Can seem scary, like an obstacle course

Very stringent requirements

Releasing your package to CRAN

“If it hurts, do it more often”



Martin Fowler: https://martinfowler.com/bliki/FrequencyReducesDifficulty.html

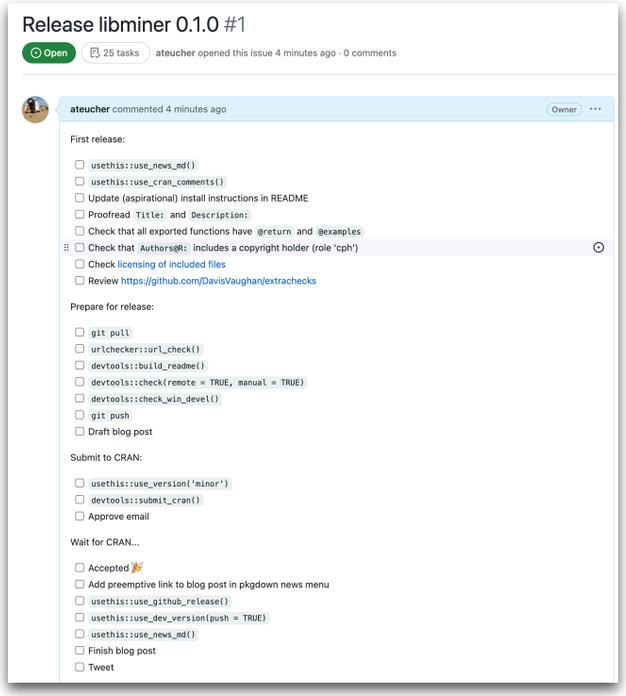
Most important advice is to build preparing for CRAN into your regular development:

- run check() often, aiming for the 3 checkmarks (0 Errors, 0 Warnings, 0 Notes)
- Use GitHub actions so R CMD check is run on multiple platforms every time you push

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



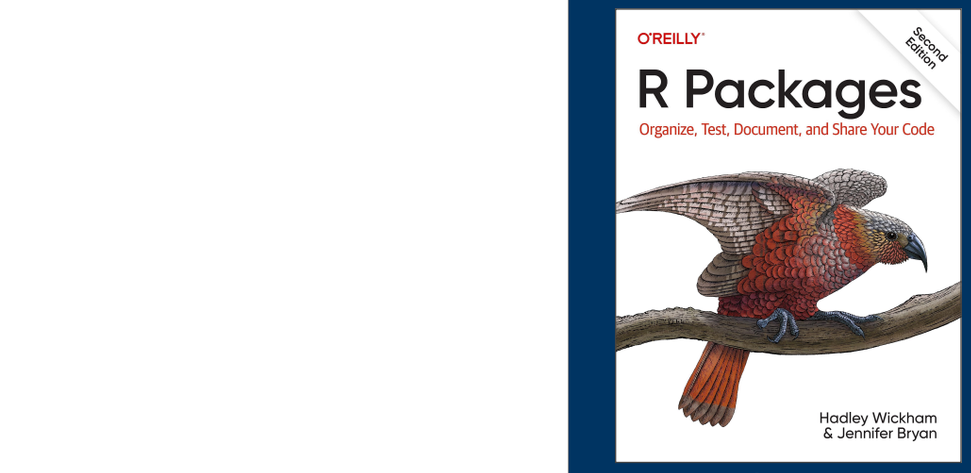
QUICK skim: pre-flight checklist, that complements your ongoing efforts to keep your package passing R CMD check cleanly and CRAN-compliant

1. Determine the release type, which dictates the version number.
2. If the package is already on CRAN: Do due diligence on existing CRAN results. If this is a first release: confirm you are in compliance with CRAN policies.
3. Freshen up documentation files, such as README.md and NEWS.md.
4. Double check() that your package is passing cleanly on multiple operating systems and on the released and development version of R.
5. Perform reverse dependency checks, if other packages depend on yours.
6. Submit the package to CRAN and wait for acceptance.
7. Create a GitHub release and prepare for the next version by incrementing the version number.
8. Publicize the new version.

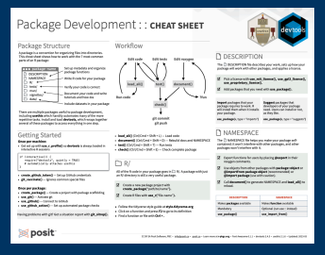
Resources

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

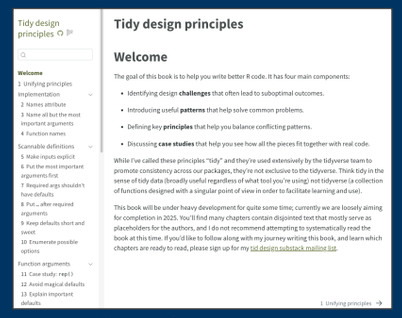
happygitwithr.com



r-pkgs.org



posit.co/resources/cheatsheets/



design.tidyverse.org
tidydesign.substack.com

Thank You!

Course Materials

pos.it/pkg-dev-conf23

Survey

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

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

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

Paste in Discord