Package Development

Angela Li
Center for Spatial Data Science
May 14, 2019

Workshop materials: bit.ly/uchipkg
Adapted from full-day workshop taught in February 2019

Make sure you have recent versions of these packages

```
packageVersion()

install.packages("devtools")
install.packages("testthat")
install.packages("usethis")
install.packages("roxygen2")
```

How is developing a package

same / different

from developing a script?

Script

One off data analysis

Primarily side-effects

Package

Defines reusable components

No side-effects

A package is a set of conventions that (with the right tools) makes your life easier

“Seriously, it doesn’t have to be about sharing your code (although that is an added benefit!). It is about saving yourself time.”

— *Hilary Parker*

How is developing a package different from writing a script?

Write functions, not “top-level” code.

Dependencies are different,
no library() calls.

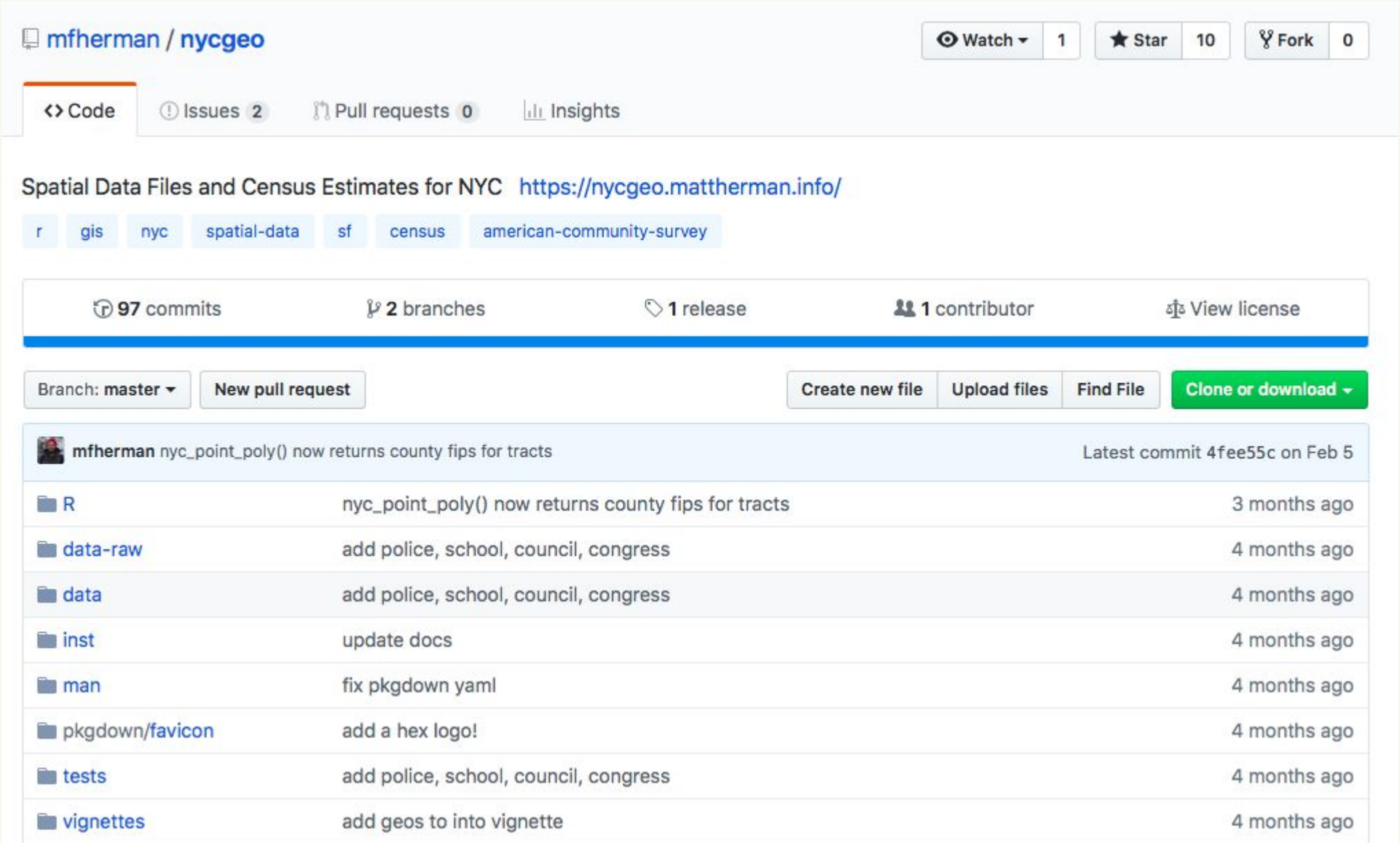
Install & Restart (or simulate that),
don’t source().

Let’s look at some packages!

Your turn

Look at this NYC shapefile package here:

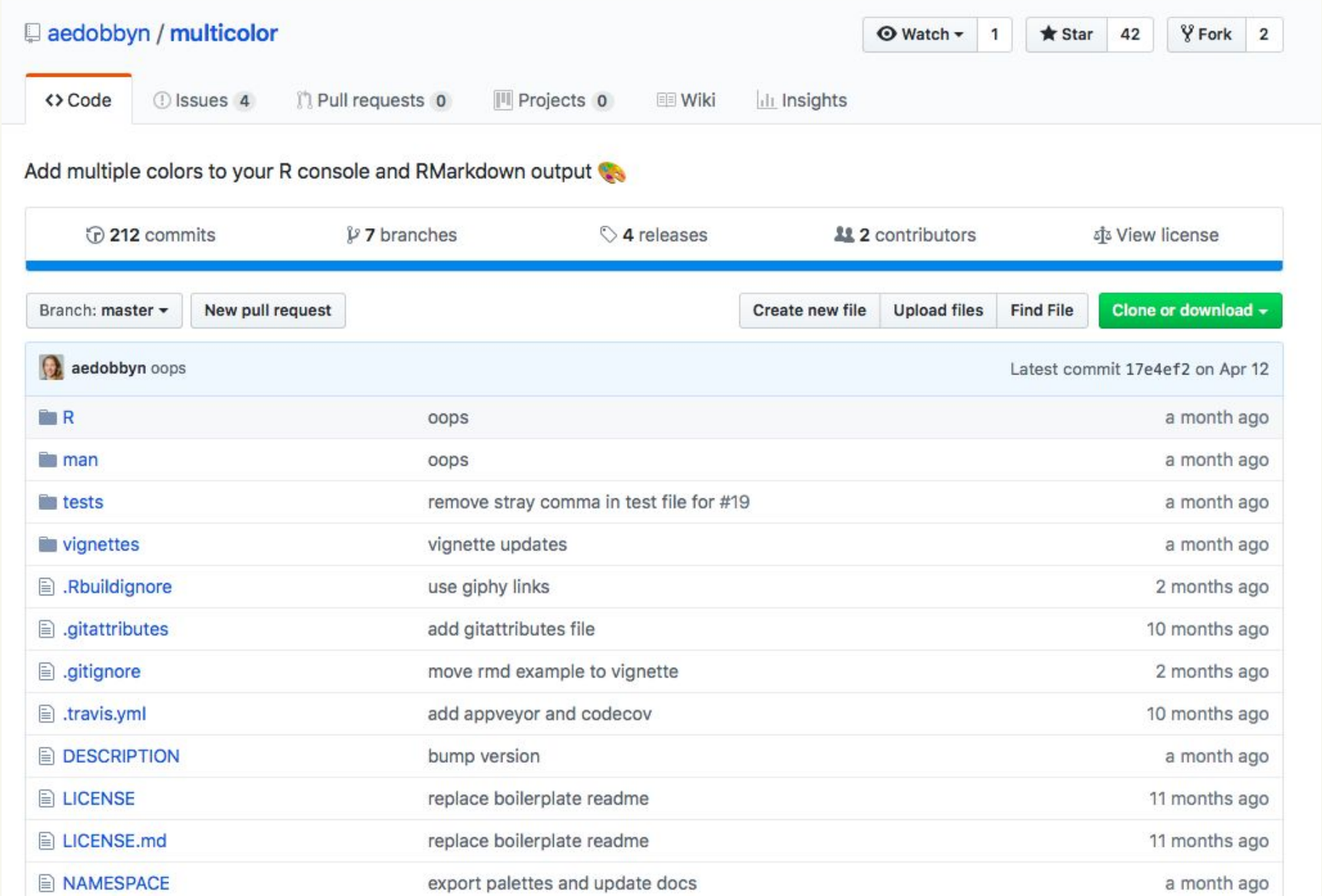
1. Where's the R code located? Which files represent data? Which files represent functions?
2. What do you think is in the DESCRIPTION file? How about the NAMESPACE file?



Your turn, pt. 2

Look at this fun text-color package here (built by a UChicago alum!):

1. What files are the same?
2. What files are different?



Major parts of a package

1. R code (“R”)
2. Documentation (“man”)
3. NAMESPACE
4. DESCRIPTION

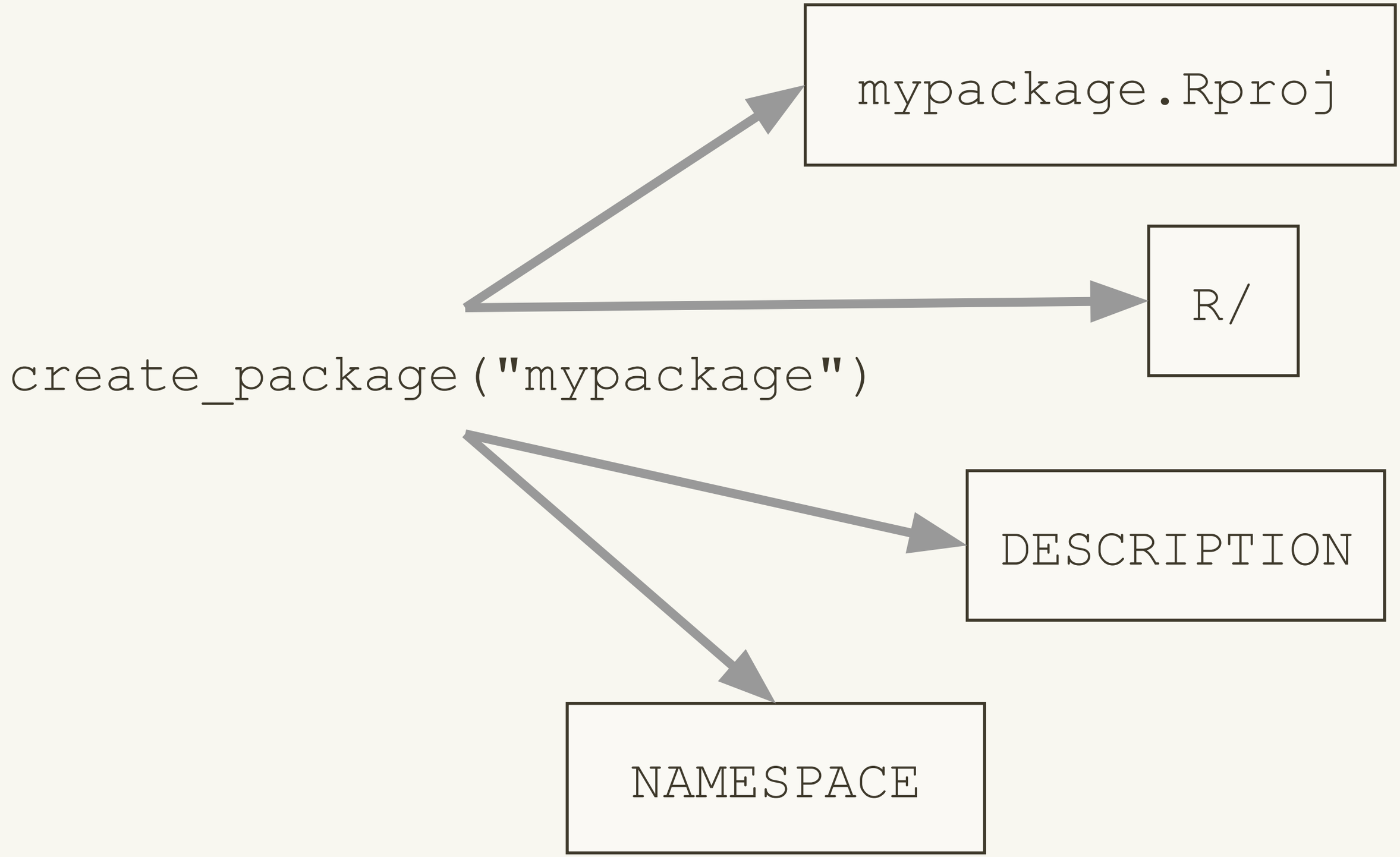
1. R code (“R”)
2. Documentation (“man”)
3. NAMESPACE
4. DESCRIPTION
5. LICENSE
6. *(recommended)* Unit tests (“tests”)

My first package

Your turn

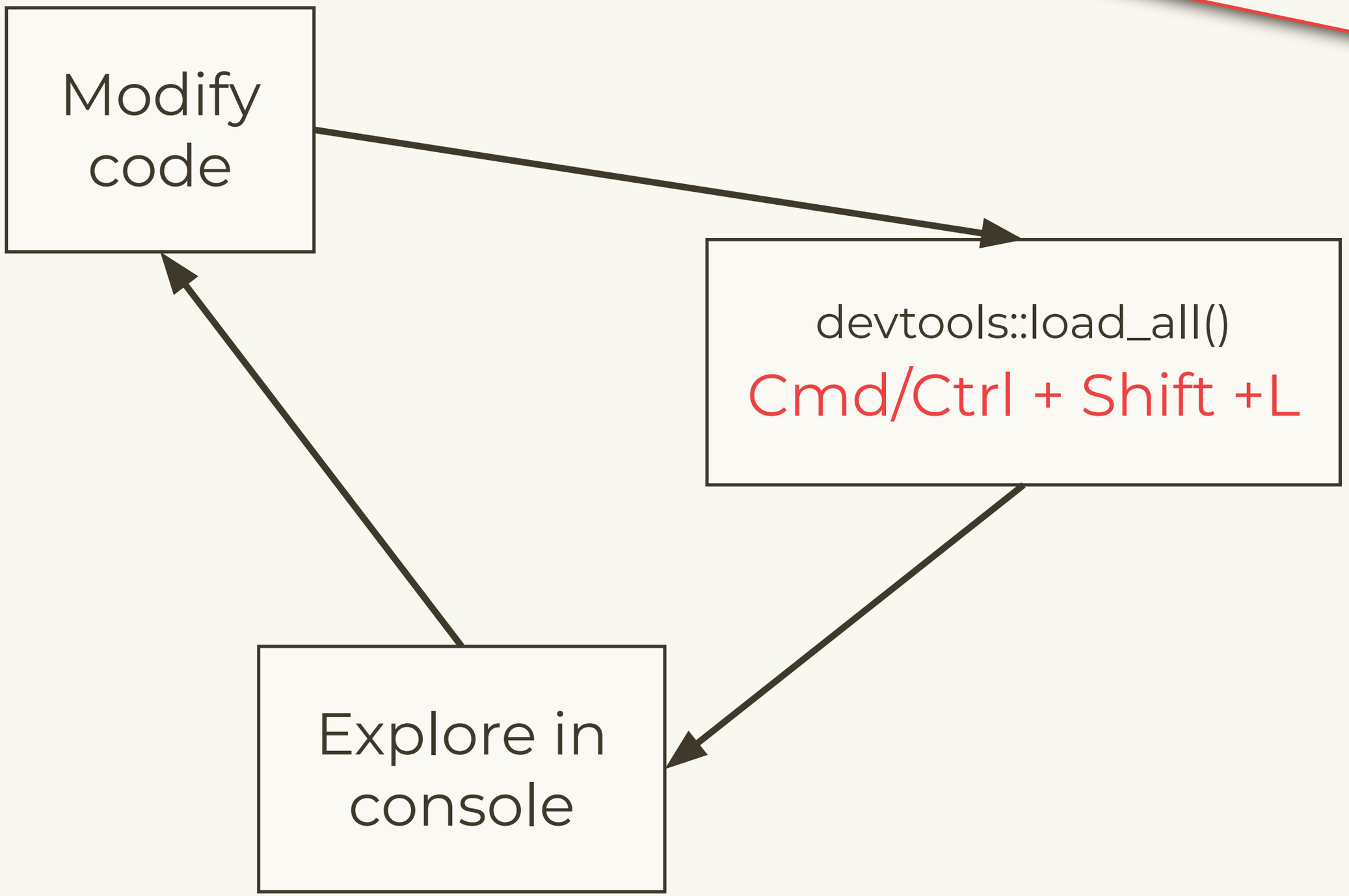
```
# Verify that you can create a package with:
usethis::create_package("~/Desktop/mypackage")

# What other files and directories are created?

# You can also create new project using RStudio
# but it has some slight differences that will
# cause hassles today (but not in general)
```

What happens we run create_package()?



Why bother?

You don't even need to save your code!



What if you need to create a new file?

```
# There's a usethis helper for that too!
usethis::use_r("file-name")

# Organize files so that related code
# lives together. If you can give a file
# a concise and informative name, it's
# probably about right
```

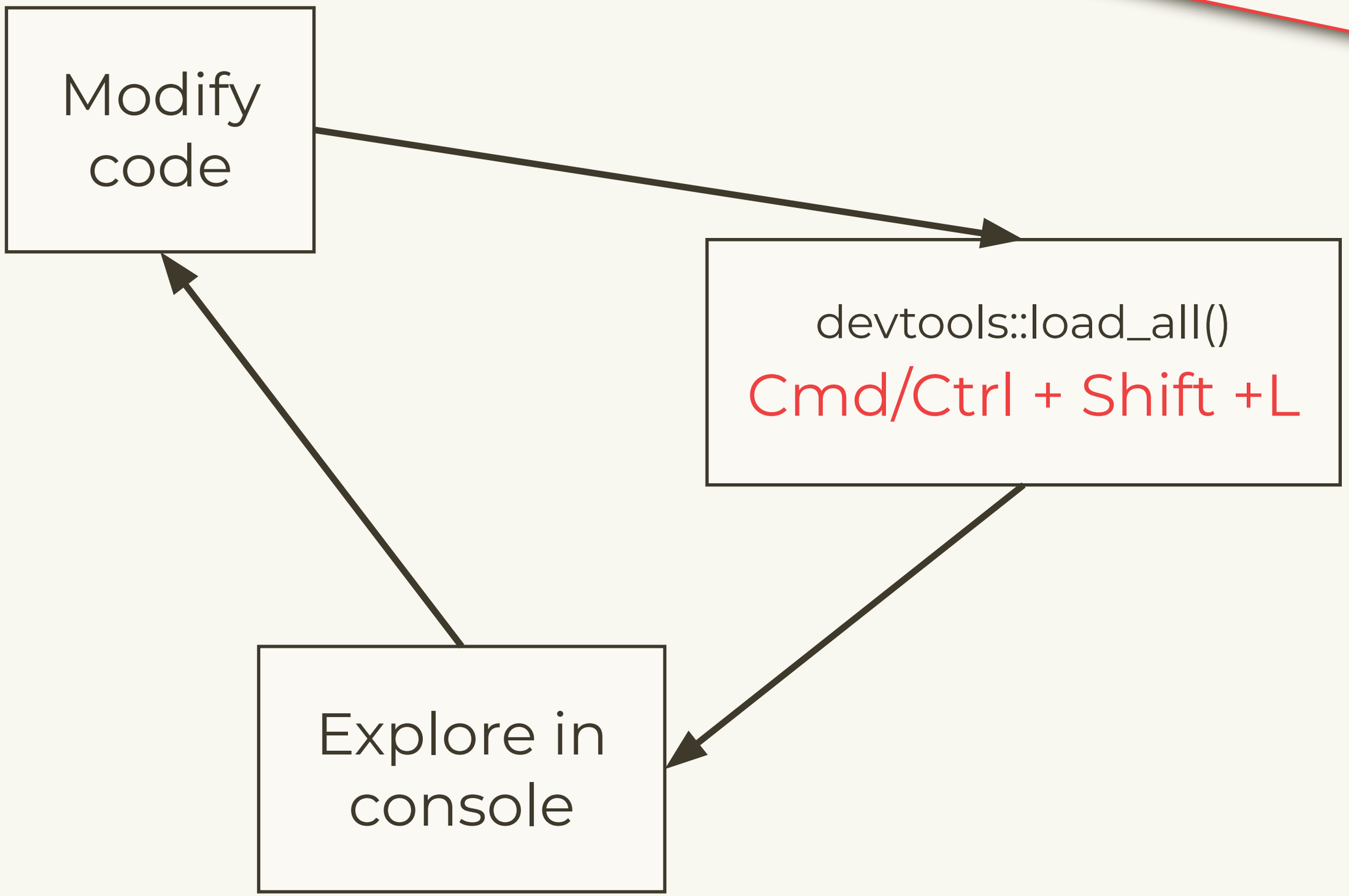
Your turn

- Create a new R file in your package called "zoo_speak.R"
  - Or use a personal function you've been wanting to put into a package!
- Paste the following code into your script:

```
zoo_speak <- function(animal, sound){
assertthat::assert_that(
  assertthat::is.string(animal),
  assertthat::is.string(sound))
  glue::glue("The ", animal, " goes ", sound,"!",
sep = " ")
  }
```

Try it out!

You don't even need to save your code!



Woohoo, you did it!

Your turn

- Change some tiny thing about your function - maybe the animal “says” instead of “goes”?
- Load all with **devtools::load_all()**

R CMD check

Automated checking

Runs automated checks for common problems in R packages.

Useful for local packages, even with some false positives.

If you want to submit to CRAN, you must pass R CMD check cleanly.

`http://r-pkgs.had.co.nz/check.html`

Types of problem

	Local	CRAN
ERROR	✓	✓
WARNING		✓
NOTE		✓

ERROR

Must fix!

WARNING

Fix if submitting
to CRAN

NOTE

Fix if submitting to CRAN

It is possible to submit with a NOTE, but
it's best avoided

Run all the checks together

```
# Cmd/Ctrl + Shift + E
devtools::check()

# If you don't understand an error,
# google it!
```

Your turn

- Check that the package is ok with **`devtools::check()`**

Looks like we get some warnings

```
── R CMD check results ──────────────────────────────────────── mypackage 0.0.0.9000 ────
Duration: 9.8s

> checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    What license it uses
  Standardizable: FALSE

> checking dependencies in R code ... WARNING
  '::' or ':::' imports not declared from:
    'assertthat' 'glue'

> checking for missing documentation entries ... WARNING
  Undocumented code objects:
    'zoo_speak'
  All user-level objects in a package should have documentation entries.
  See chapter 'Writing R documentation files' in the 'Writing R
  Extensions' manual.

0 errors ✔ | 3 warnings ✖ | 0 notes ✔
```

Looks like we get some warnings

```
── R CMD check results ──────────────────────────────────── mypackage 0.0.0.9000 ────
Duration: 9.8s

❯ checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    What license it uses
  Standardizable: FALSE

❯ checking dependencies in R code ... WARNING
  '::' or ':::' imports not declared from:
    'assertthat' 'glue'

❯ checking for missing documentation entries ... WARNING
  Undocumented code objects:
    'zoo_speak'
  All user-level objects in a package should have documentation entries.
  See chapter 'Writing R documentation files' in the 'Writing R
  Extensions' manual.

0 errors ✔ | 3 warnings ✖ | 0 notes ✔
```

Fix DESCRIPTION

Looks like we get some warnings

```
── R CMD check results ──────────────────────────────────────── mypackage 0.0.0.9000 ────
Duration: 9.8s

❯ checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    What license it uses
  Standardizable: FALSE

❯ checking dependencies in R code ... WARNING
  '::' or ':::' imports not declared from:
    'assertthat' 'glue'

❯ checking for missing documentation entries ... WARNING
  Undocumented code objects:
    'zoo_speak'
  All user-level objects in a package should have documentation entries.
  See chapter 'Writing R documentation files' in the 'Writing R
  Extensions' manual.

0 errors ✔ | 3 warnings ✖ | 0 notes ✔
```

Fix DESCRIPTION

Declare dependencies

Looks like we get some warnings

```
── R CMD check results ──────────────────────────────────────── mypackage 0.0.0.9000 ────
Duration: 9.8s

> checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    What license it uses
  Standardizable: FALSE

> checking dependencies in R code ... WARNING
  '::' or ':::' imports not declared from:
    'assertthat' 'glue'

> checking for missing documentation entries ... WARNING
  Undocumented code objects:
    'zoo_speak'
  All user-level objects in a package should have documen
  See chapter 'Writing R documentation files' in the 'Writ
  Extensions' manual.

0 errors ✔ | 3 warnings ✖ | 0 notes ✔
```

Fix DESCRIPTION

Declare dependencies

Document our package

Documentation!

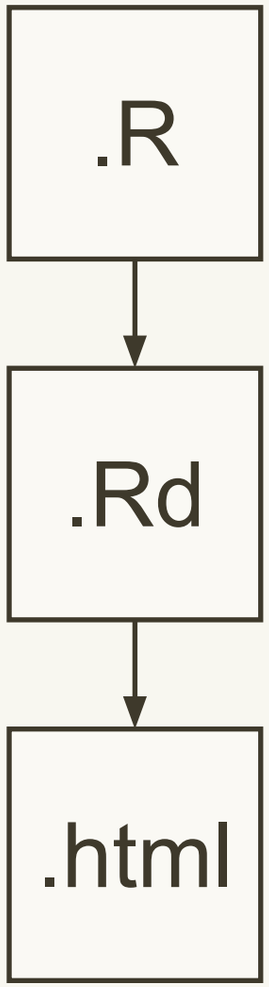
Why?

People need instructions to use new things!

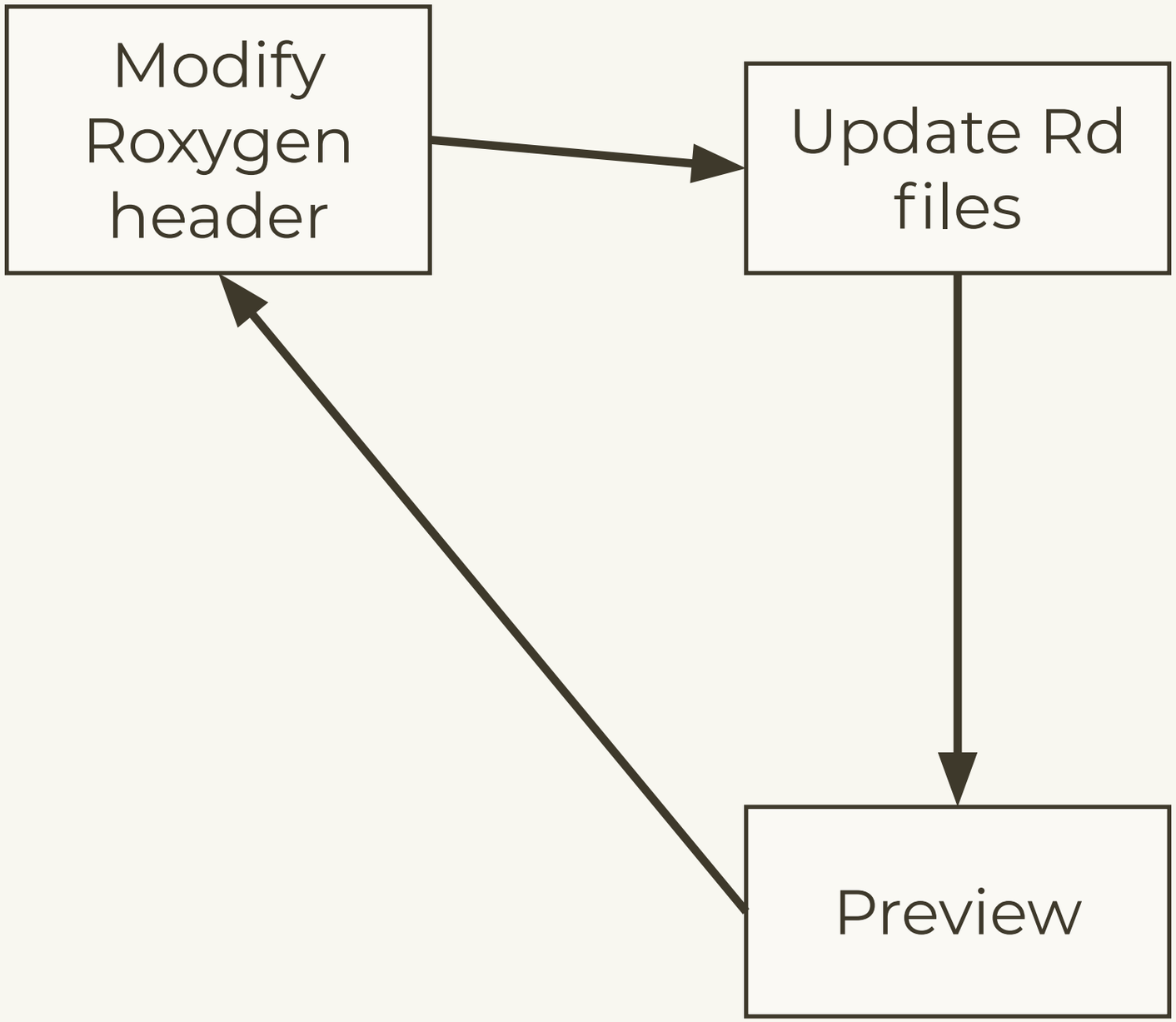
You might want instructions to remind you how your tools work too.

Documentation is the way you preserve the information about your tools.

Function-level with roxygen2



Documentation workflow

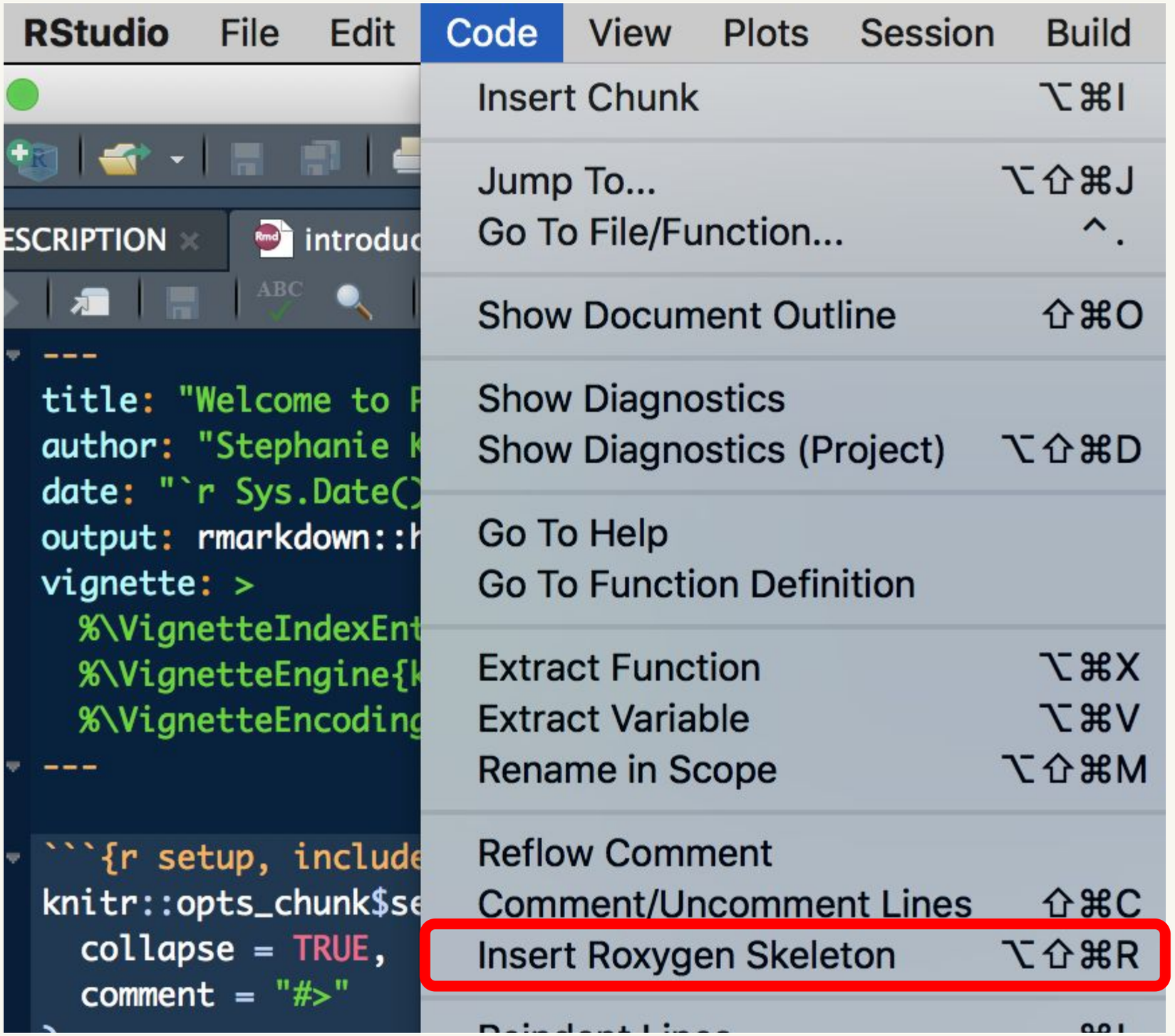


Cmd/Ctrl + Shift + D

devtools::document()

?topicname

Modify Roxygen header



Roxygen allows us to explain the function's parts...

http://r-pkgs.had.co.nz/man.html

Your turn

Open up a .R function file from the provided examples: multi_color.R or nyc_boundaries.R.

Does it have a Roxygen header?
If so, can you find a parameter and read the documentation?

Can you find an example of how to use the function?

Roxygen renders to .Rd in /man/ folder

In almost all cases you can ignore these files

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/add_col.R
\name{add_col}
\alias{add_col}
\title{Add a column to a data frame}
\usage{
add_col(x, name, value, where = -1)
}
\arguments{
\item{x}{A data frame}

\item{name}{Name of variable to create. If a variable of that
name
already exists it will be replaced}

\item{value}{Values to insert.}
```

add_col {hadcol}

Add a column to a data frame

R translates .Rd into .html for viewing

Description

Allows you to specify the position. Will replace existing variable with the same name if present.

Usage

```
add_col(x, name, value, where = -1)
```

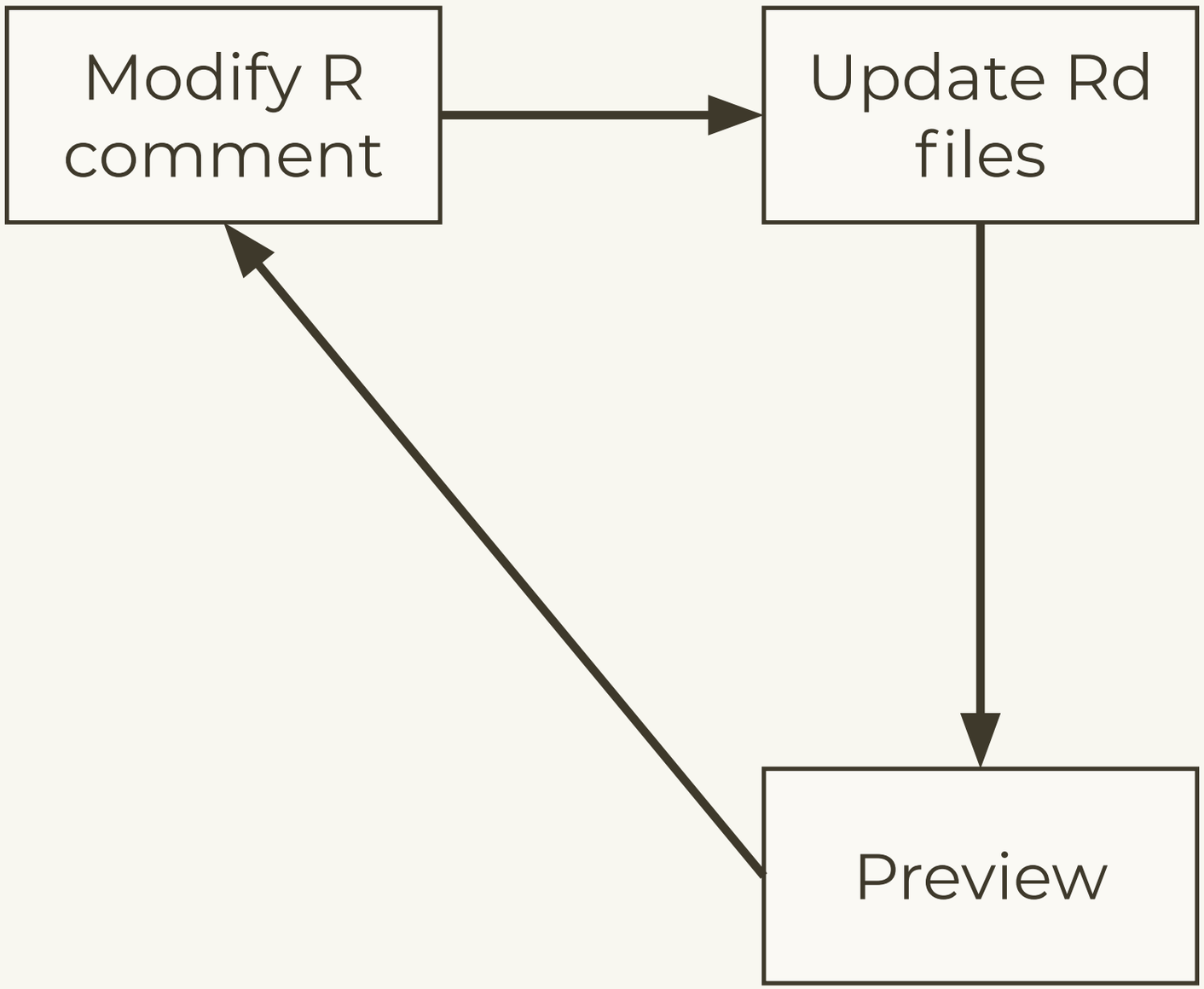
Arguments

`x`	A data frame
`name`	Name of variable to create. If a variable of that name already exists it will be replaced
`value`	Values to insert.
`where`	Position to insert. Use 1 to insert on LHS, or -1 to insert on RHS.

Examples

```
df <- data.frame(x = 1:5)
add_col(df, "y", runif(5))
add_col(df, "y", runif(5), where = 1)

add_col(df, "x", 5:1)
```

Documentation workflow



Cmd/Ctrl + Shift + D

devtools::document()

?topicname

Your turn

Now, open the .R file for your zoo_speak() function and:

- add a Roxygen header (Code > Insert Roxygen skeleton)
  - include @param, @return, and @examples
- run **devtools::document()** to produce zoospeak.Rd

Read online about how to document other objects

Data

https://r-pkgs.org/data.html#documenting-data

Classes & methods

https://r-pkgs.org/man.html#man-classes

Packages

https://r-pkgs.org/man.html#man-packages

Your turn

- Check the package again with **`devtools::check()`**

Two warnings to go!

```
── R CMD check results ──────────────────────────────────────────
Duration: 14s

> checking DESCRIPTION meta-information ...
  Non-standard license specification:
    What license it uses
  Standardizable: FALSE

> checking dependencies in R code ... WARN
  '::' or ':::' imports not declared from:
    'assertthat' 'glue'

0 errors ✔ | 2 warnings ✖ | 0 notes ✔
```

Fix DESCRIPTION

Declare dependencies

Dependencies

```r
# use_package() will modify the DESCRIPTION
# and remind you how to use the function.
usethis::use_package("assertthat")
usethis::use_package("glue")
```

```
library(xyz)
require(xyz)
```

I need you!

```
Depends:
  R (>= 3.0.2) # optional version spec
Imports:
  stringr (>= 1.0.0),
  lubridate
Suggests:
  ggplot2
```

I like having you around

There are three types of dependency

Imports = required**.** Installed automatically.

Suggests = optional: development only; used in vignette or example. Not installed automatically.

Depends = basically deprecated for packages. (Correct uses exist, but beyond the scope of this class)

Reasons to use depends instead of imports

This page has been intentionally left blank

Your turn

- Add “assertthat” and “glue” using **usethis::use_package()**

- Load all with **devtools::load_all()**
- Check the package again with **devtools::check()**

One last warning

```
── R CMD check results ──────────────────────────────────── mypackage 0.0.0.9000 ────
Duration: 14.8s

> checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    What license it uses
  Standardizable: FALSE

0 errors ✔ | 1 warning ✖ | 0 notes ✔
```

Fix DESCRIPTION

License

There are three main open source licenses

CC0

"public domain", best for data packages

MIT

Free for anyone to do anything with

GPL

Changes and bundles must also be GPL

These are gross simplifications!

Use helper to set up

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

You can also make clear that your package isn't open source

DESCRIPTION:
License: file LICENSE

LICENSE:
Proprietary: do not distribute outside of Widgets Incorporated.

Your turn

- Add an open source license with “use_mit_license(name = “Your Name”)

- Load all with **devtools::load_all()**
- Check the package again with **devtools::check()**

We're done!

```
── R CMD check results ──────────────────────────────────
Duration: 18.7s

0 errors ✔ | 0 warnings ✔ | 0 notes ✔
```

A few other things...

Vignettes

Easiest way to get started is with use_vignette()

```
usethis::use_vignette("name")

# Adds to DESCRIPTION
Suggests: knitr
VignetteBuilder: knitr

# Creates vignettes/
# Drafts vignettes/name.Rmd
```

README

If sharing with others, include a README

```
Your choice: but often useful to include
results of running code
usethis::use_readme_md()
usethis::use_readme_rmd()
```

For public projects this should include a brief overview, instructions on how to install, and a few examples. For private projects, this is a great place to jot down notes!

NEWS

Also good idea to track changes

```
usethis::use_news_md()
```

Namespace: imports

You might get tired of using :: all the time

```
# Or you might want to use an infix
function
`%>%` <- magittr::`%>%`

col_summary <- function(df, fun) {
  stopifnot(is.data.frame(df))

  df %>%
    purrr::keep(is.numeric) %>%
    purrr::modify(fun)
}
```

You can **import** functions into the package

```
#' @importFrom purrr keep modify
#' @importFrom magrittr %>%
col_summary <- function(df, fun) {
  stopifnot(is.data.frame(df))

  df %>%
    keep(is.numeric) %>%
    modify(fun)
}
```

Alternatively, create R/imports.R

```
# Imports belong to the package, not to
# individual functions, so you might want
# to recognise this by storing in a central
# location

#' @importFrom purrr keep map
#' @importFrom magrittr %>%
NULL
```

Importing everything from a package seems easy

```
#' @import purrr
col_summary <- function(df, fun) {
  stopifnot(is.data.frame(df))

  df %>%
    keep(is.numeric) %>%
    map_dfc(fun)
}
```

But is dangerous...

```
#' @import foo
#' @import bar
fun <- function(x) {
  fun1(x) + fun2(x)
}

# Works today
# But next year, bar package adds fun1
function
```

Description	NAMESPACE
Makes **package** available	Makes **function** available
Mandatory	Optional (can use :: instead)
use_package()	#' @importFrom

Namespace: exports

A namespace splits functions into two classes

Internal	**External**
Only for use within package	For use by others
Documentation optional	Must be documented
Easily changed	Changing will break other people’s code

The default NAMESPACE exports everything

```
# Generated by roxygen2: fake comment so
# roxygen2 overwrites silently.
exportPattern("^[^\\.]")
```

Better to export function explicitly

```
#' @export
fun1 <- function(...) {}

#' @export
fun2 <- function(...) {}
```

Most important if you're planning on sharing with others

Export functions that people should use

```
# Don't export internal helpers

# Defaults for NULL values
`%||%` <- function(a, b) if (is.null(a)) b
else a

# Remove NULLs from a list
compact <- function(x) {
  x[!vapply(x, is.null, logical(1))]
}
```

Git + GitHub

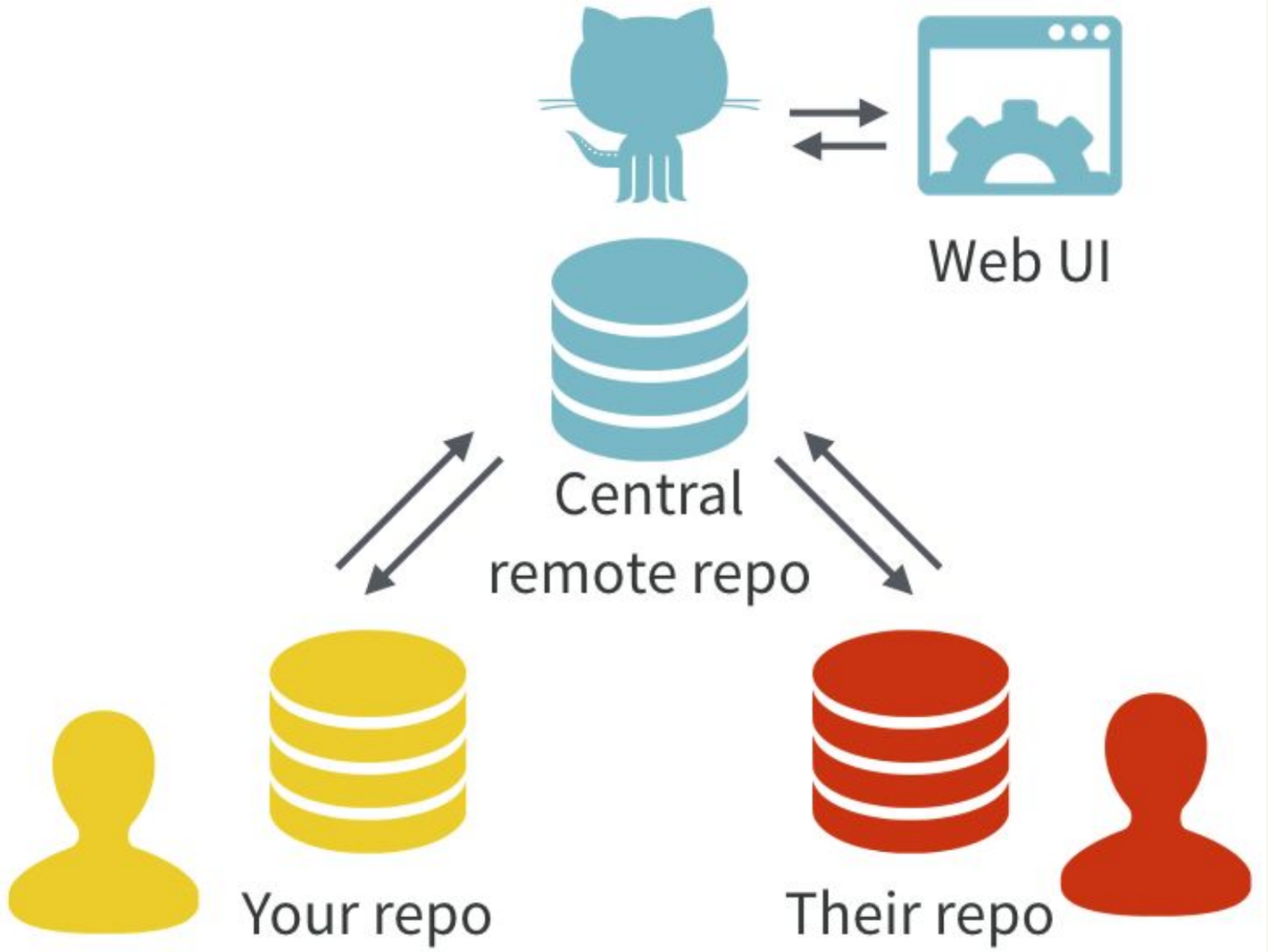
Use both!

The 😡💩😱 will pay off

happygitwithr.com

Excuse me, do you have a moment to talk about version control?

https://doi.org/10.7287/peerj.preprints.3159v2



Testing Workflow

http://r-pkgs.had.co.nz/tests.html

Our Silly Example Function

zooSounds.R

```
goToTheZoo <- function(animal, sound){
assertthat::assert_that(
  assertthat::is.string(animal),
  assertthat::is.string(sound))
  glue::glue("The ", animal, " goes ", sound,"!",
sep = " ")
}
```

Even more convenient with some conventions

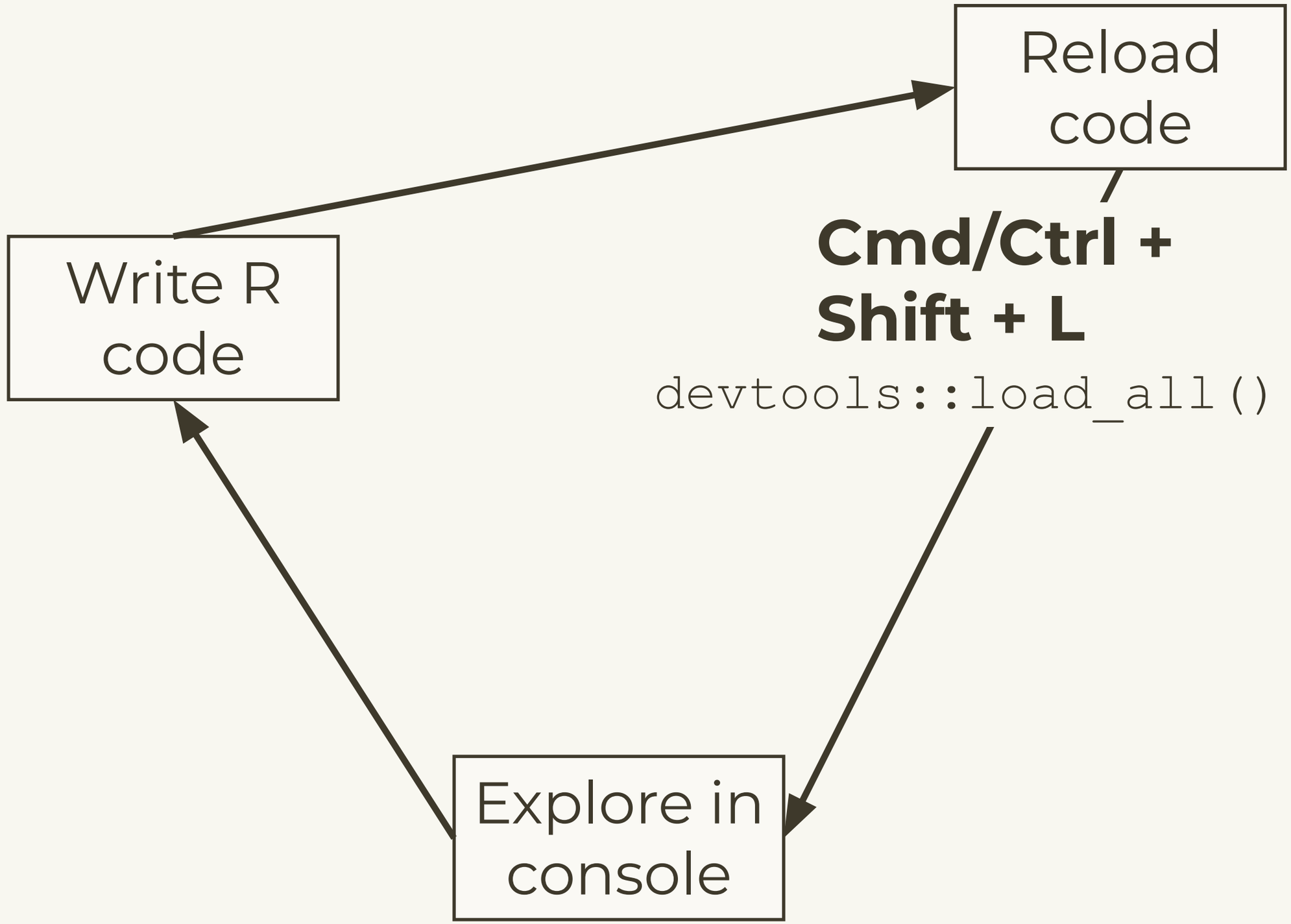
Set up testthat infrastructure

```
usethis::use_test()
✔ Adding 'testthat' to Suggests field
✔ Creating 'tests/testthat/'
✔ Writing 'tests/testthat.R'
✔ Writing 'tests/testthat/test-zooSounds.R'
● Modify 'tests/testthat/test-zooSounds.R'
```

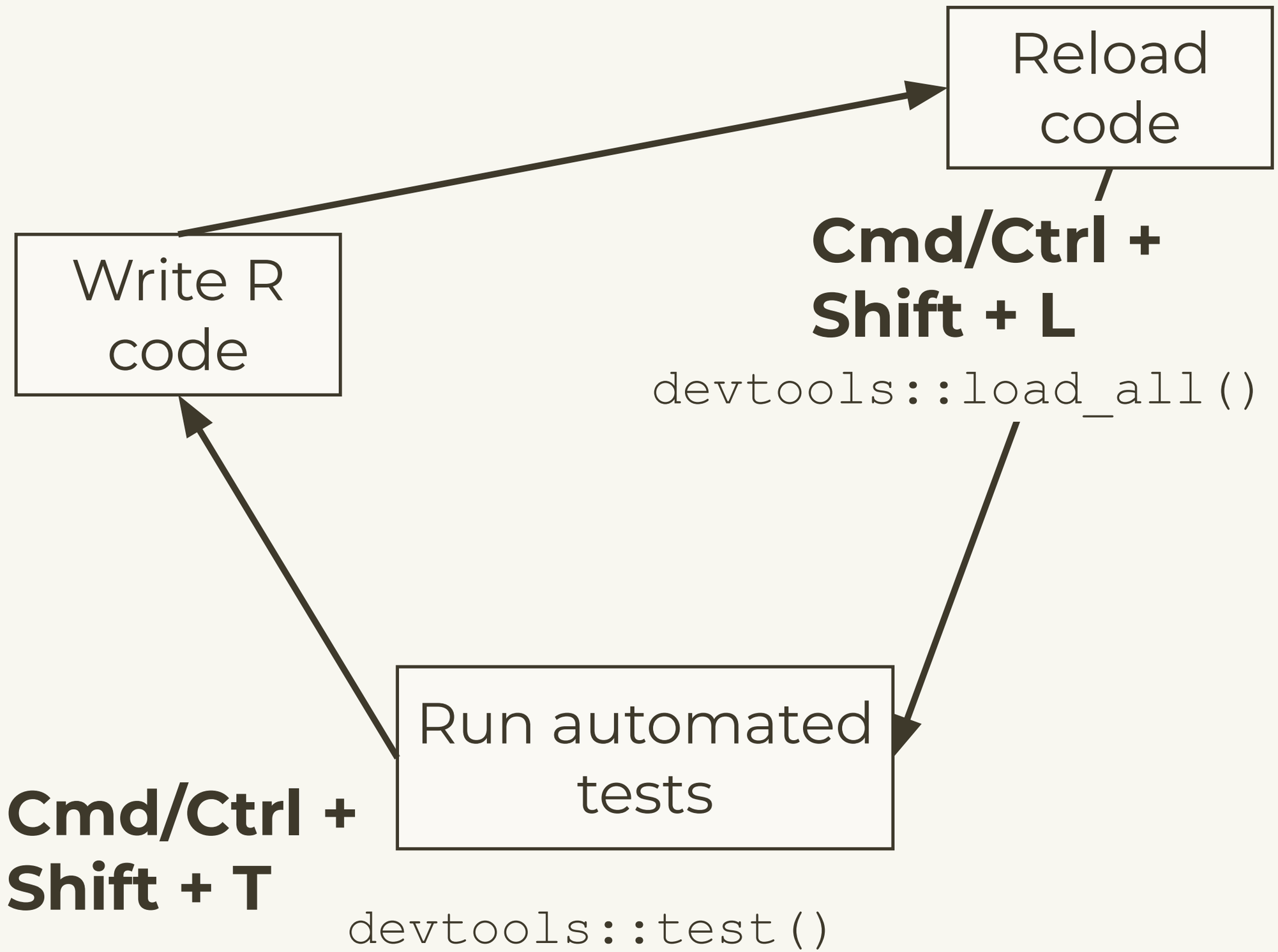
```
devtools::test()
# Or Command + Shift + T
```

Create test file matching script

So far we've done this:



Testthat gives a new workflow



A sample test

Tests for R/zooSounds.R

```
# In tests/testthat/test-zooSounds.R

library(testthat)

test_that("goToTheZoo produces expected strings", {
  allSounds <- as.character(goToTheZoo("giraffe",
"moo"))
  expect_equal(allSounds, "The giraffe goes moo!")
})

test_that("goToTheZoo fails with numbers", {
  expect_error(goToTheZoo(1, 2))
})
```

Four expectations cover 90% of cases

```
expect_equal(obj, exp)

expect_error(code, regexp)

expect_warning(code, regexp)
expect_warning(code, NA)

expect_known_output(code)
```

Organizing Tests

Think about the overall functionality, or “end to end” tests

Test every individual task the function completes separately

Check both for successful situations and for expected failure situations

Practice the workflow

- Copy in your `goToTheZoo()` function.
- Create a `goToTheZoo()` test file using `use_test()`
- Put the previous expectations in a test case.
- Verify that the tests pass with `Cmd + Shift +T`.
- Add test using `animal = 7`. Verify that it fails.

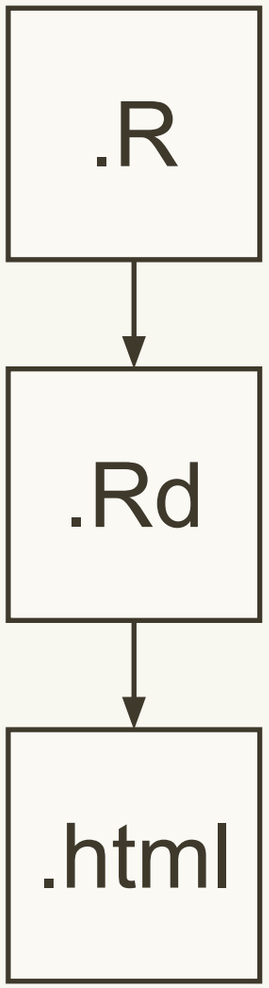
Why?

People need instructions to use new things!

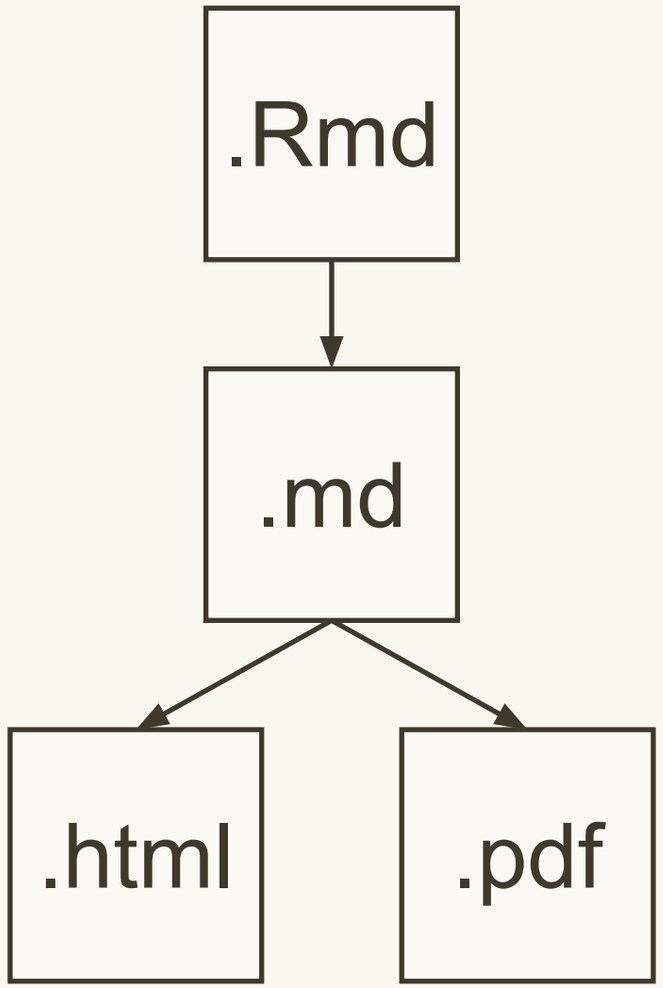
You might want instructions to remind you how your tools work too.

Documentation is the way you preserve the information about your tools.

Function-level with roxygen2



Package-level with rmarkdown

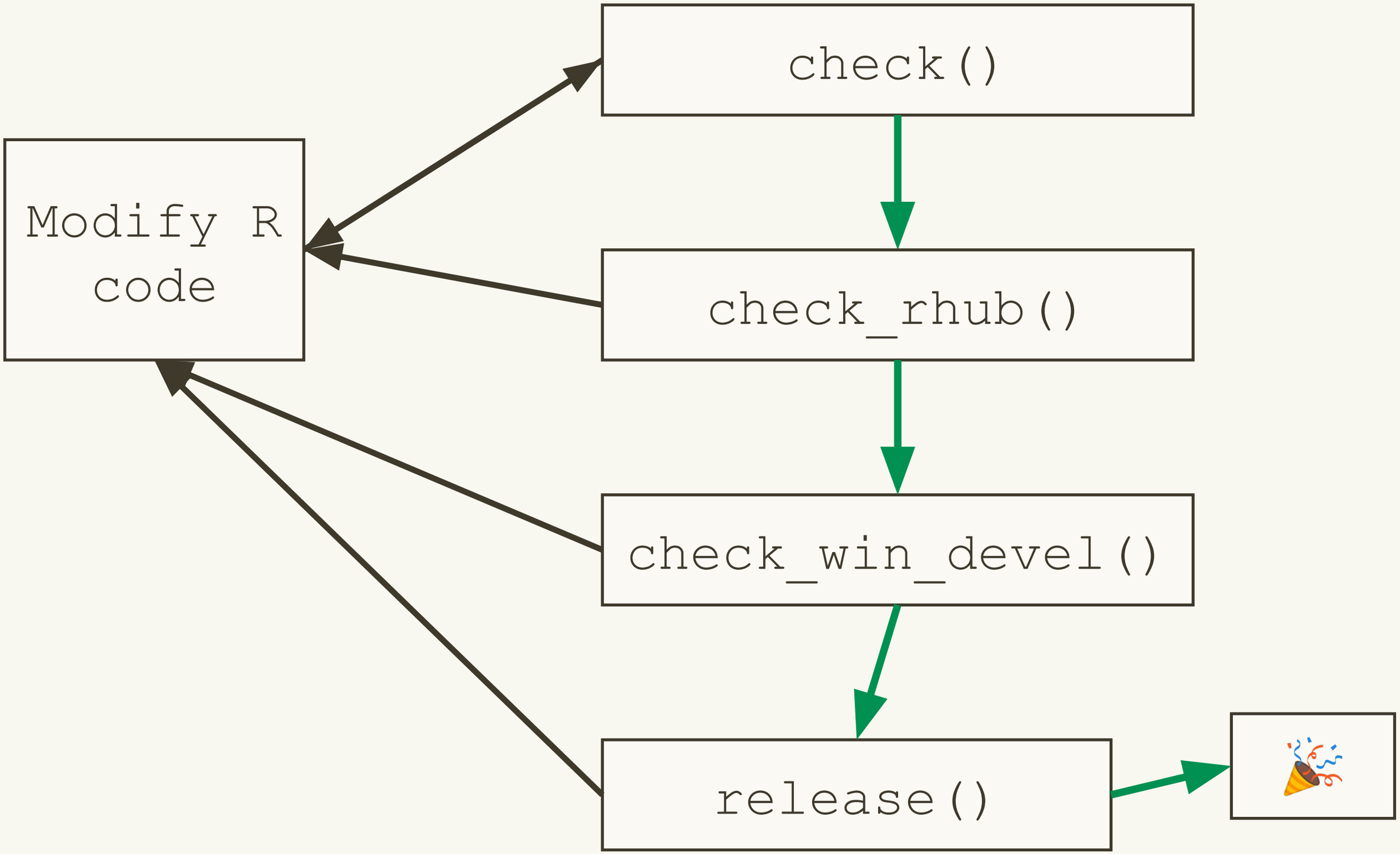


CRAN

(varsity level)

First submission to CRAN

```r
# First check locally
devtools::check()

# Then on R-hub
devtools::check_rhub()

# Then with CRAN's win-builder
devtools::check_win()

# Write submission notes
usethis::use_cran_comments()

# Then submit to CRAN
devtools::release()
# This asks you questions which you should
# carefully read and answer
```


check()
Modify R code
check_rhub()
check_win_devel()
release()

cran-comments.md

Goal is to illustrate that you've done your due diligence

```
## Test environments
* local OS X install (R-release)
* win-builder (R-release, R-devel)

## R CMD check results

0 errors | 0 warnings | 1 note

* This is a new release.
```

There's always one note for a new submission

If your submission fails

Do not despair! It happens to everyone, even R-core members.

If it’s from the CRAN robot, just fix the problem & resubmit.

If it’s from a human, do not respond to the email and do not argue. Instead update cran-comments.md & resubmit.

For resubmission:

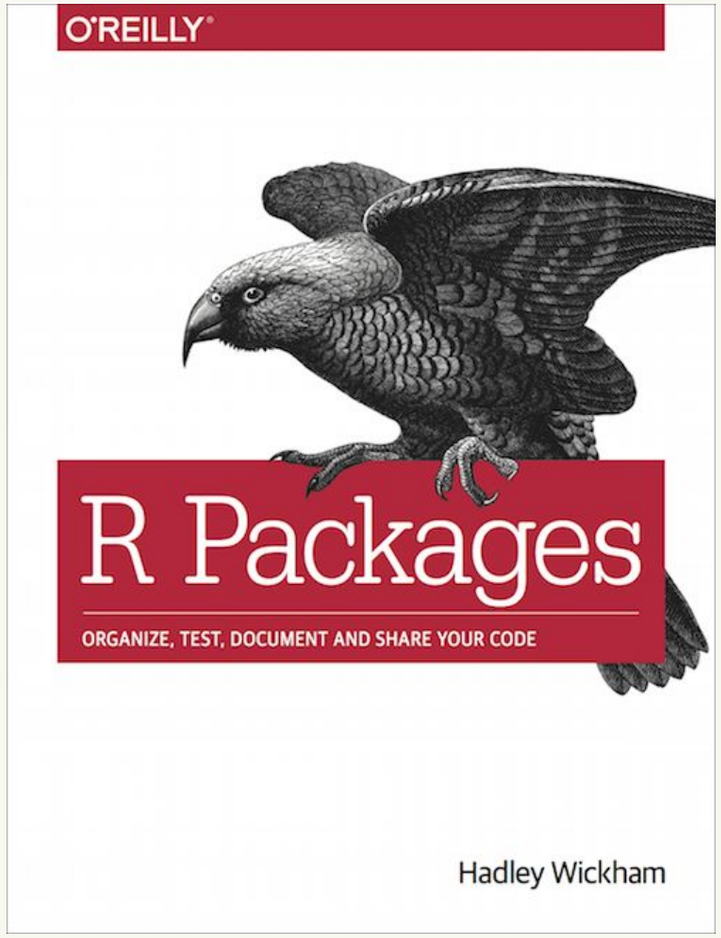
```
This is a resubmission. Compared to the last
submission, I have:

* First change.
* Second change.
* Third change

---
## Test environments
* local OS X install, R 3.2.2
* win-builder (devel and release)

## R CMD check results

...
```

Subsequent submissions to CRAN

```
# Proceed as before. If you have reverse
dependencies
# you need to also run R CMD check on them, and
# notify CRAN if you have deliberately broken
them.
# Fortunately the revdepcheck package makes this
# fairly easy

install_github("r-lib/revdepcheck")

use_revdep_check()

library(revdepcheck)
revdep_check()
revdep_report_cran()
```

Learning more

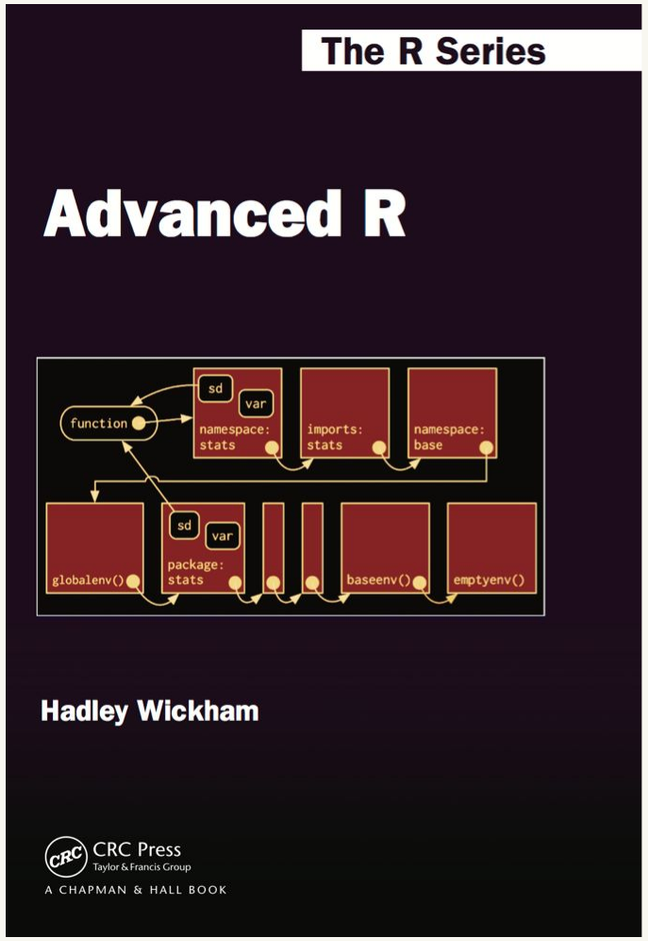
Read “The Whole Game” in *R Packages*



https://r-pkgs.org/whole-game.html

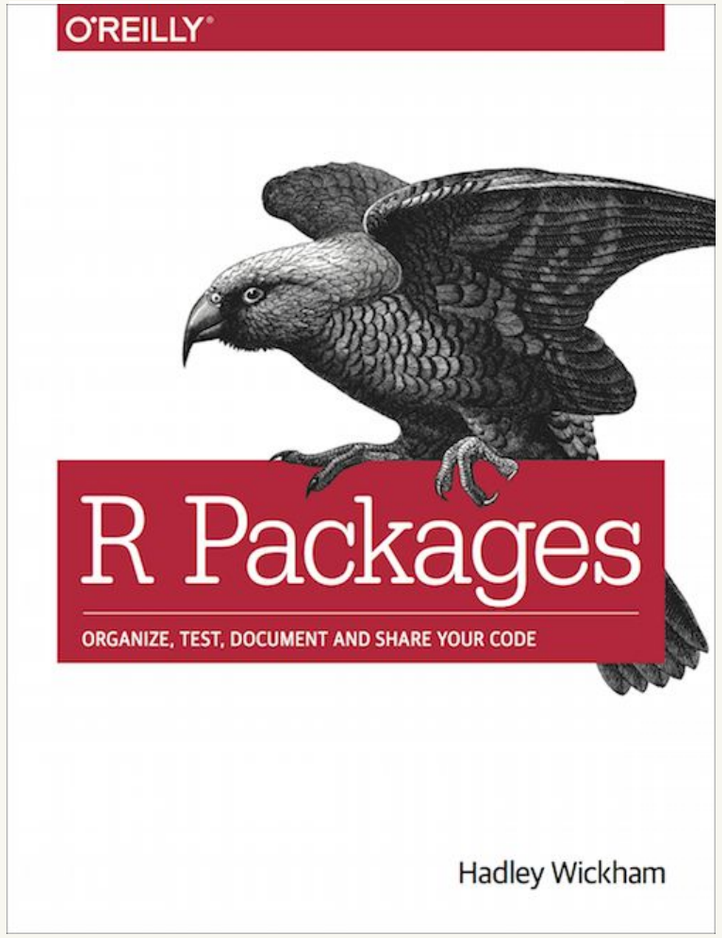
https://amzn.com/1491910399

More details on many topics in books



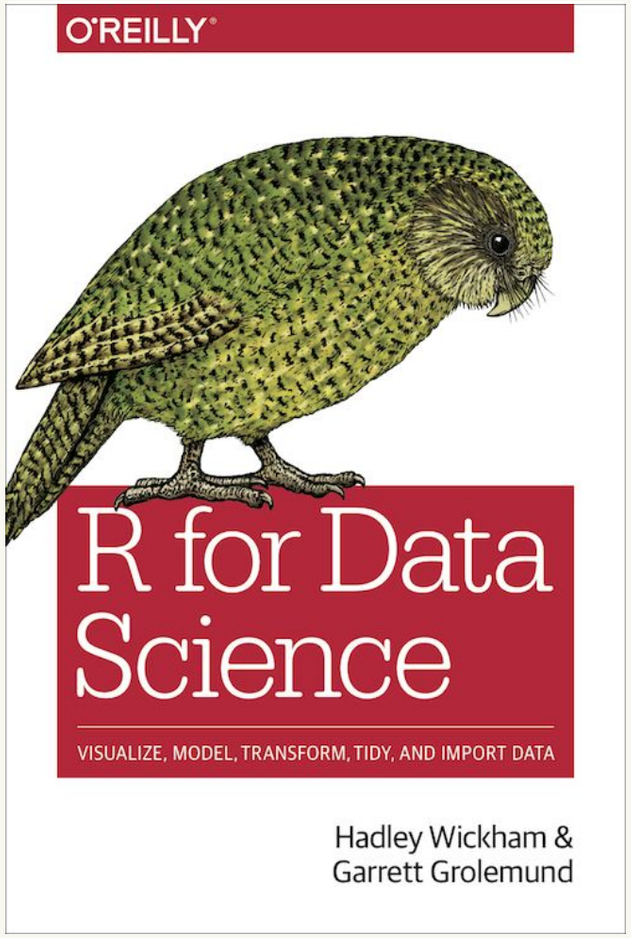
http://adv-r.hadley.nz/

http://amzn.com/1466586966



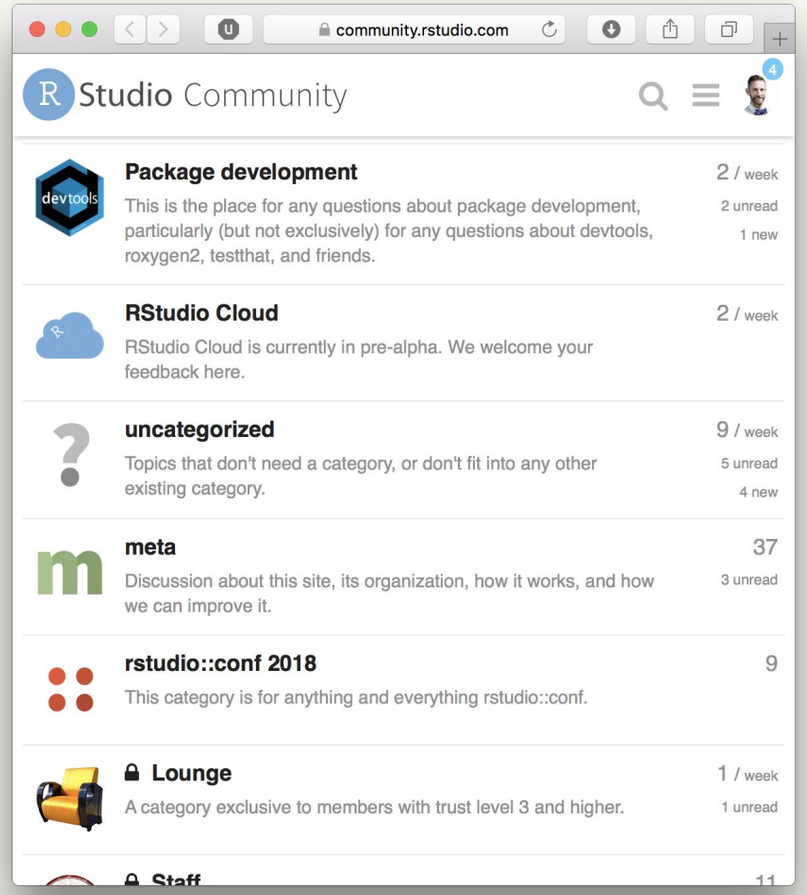
http://r-pkgs.org

https://amzn.com/1491910399

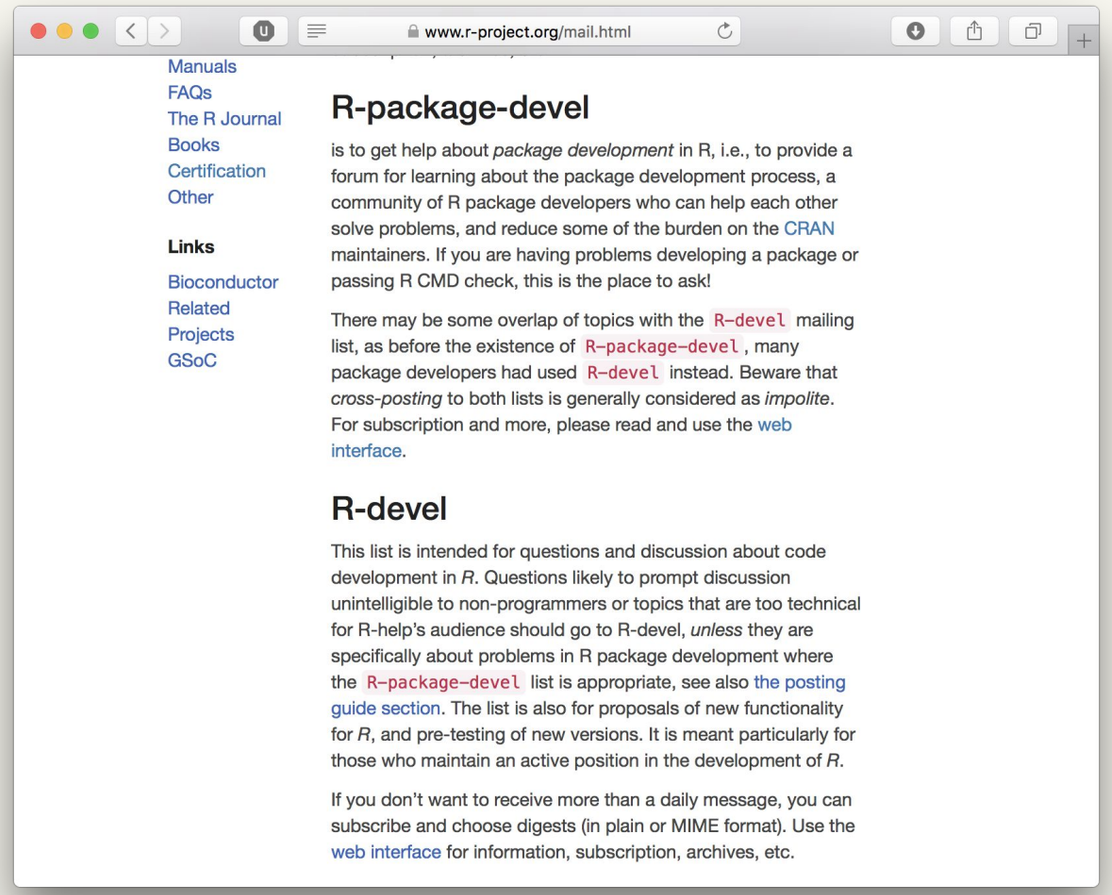


http://r4ds.had.co.nz

https://amzn.com/1491910399

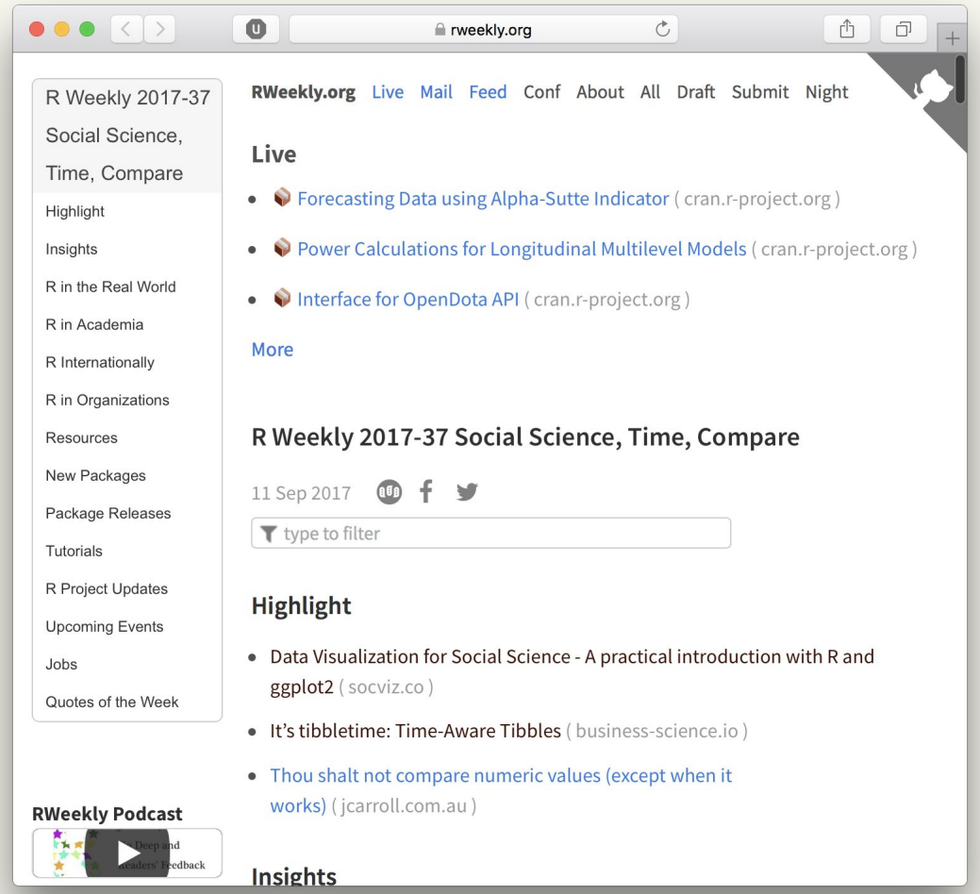


community.rstudio.com

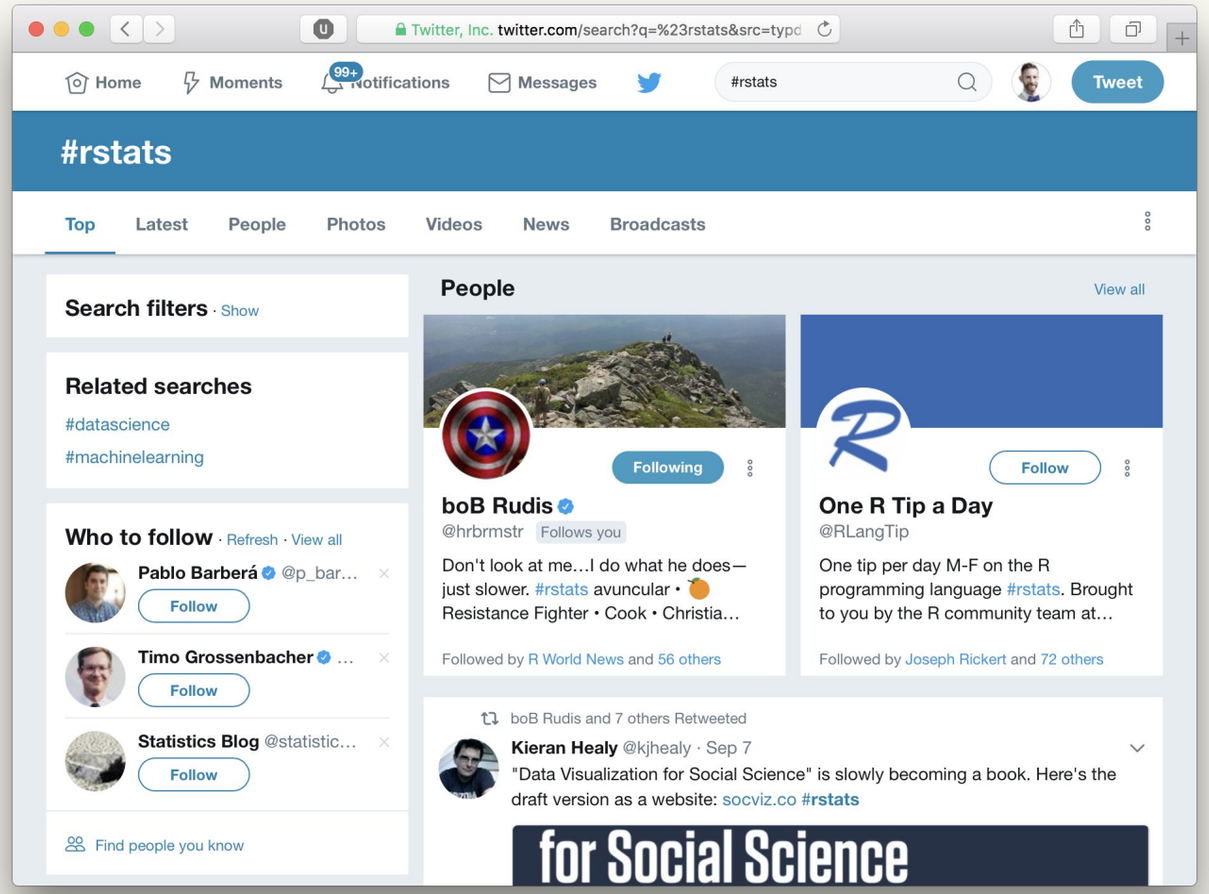


R-package-devel mailing list

rweekly.org



#rstats



stackoverflow

[r] score:5 is:question closed:no