Packages & R Code

September 2019

Angela Li
Samantha Toet

Workshop materials: bit.ly/cville_pkg

Motivation

“Workflow: you should have one”

— *Jenny Bryan*

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*

Script

One off data analysis

Primarily side-effects

Package

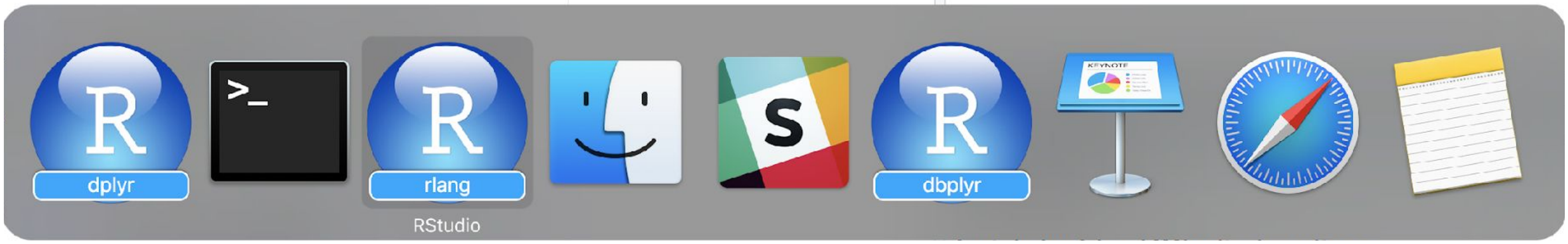
Defines reusable components

No side-effects

RStudio Projects

Why use RStudio projects?

3 reasons



Work on multiple projects simultaneously and independently

Manage working directories

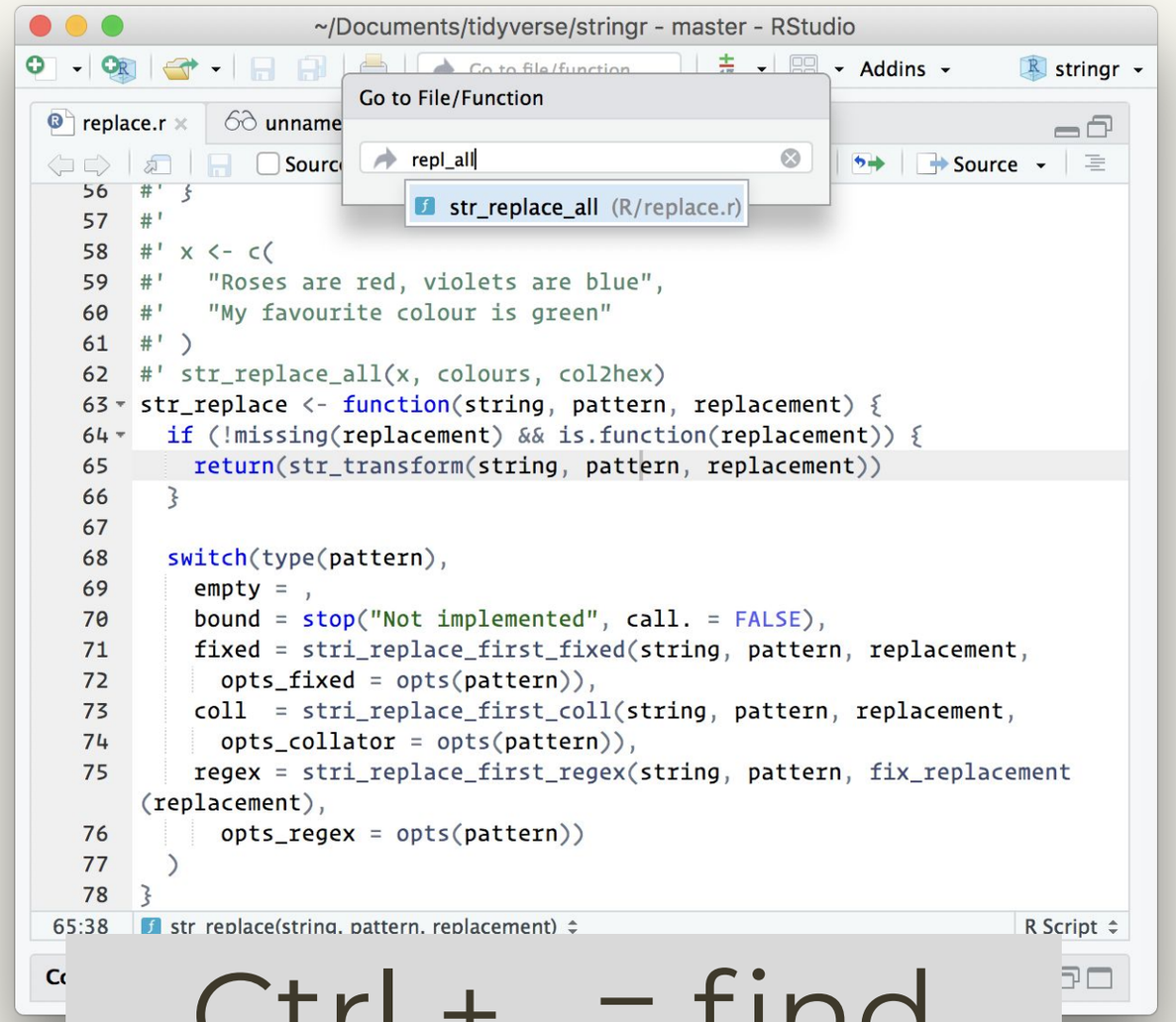
If the first line of your #rstats script is

```
setwd("C:\Users\jenny\path\that\only\I\have")
```

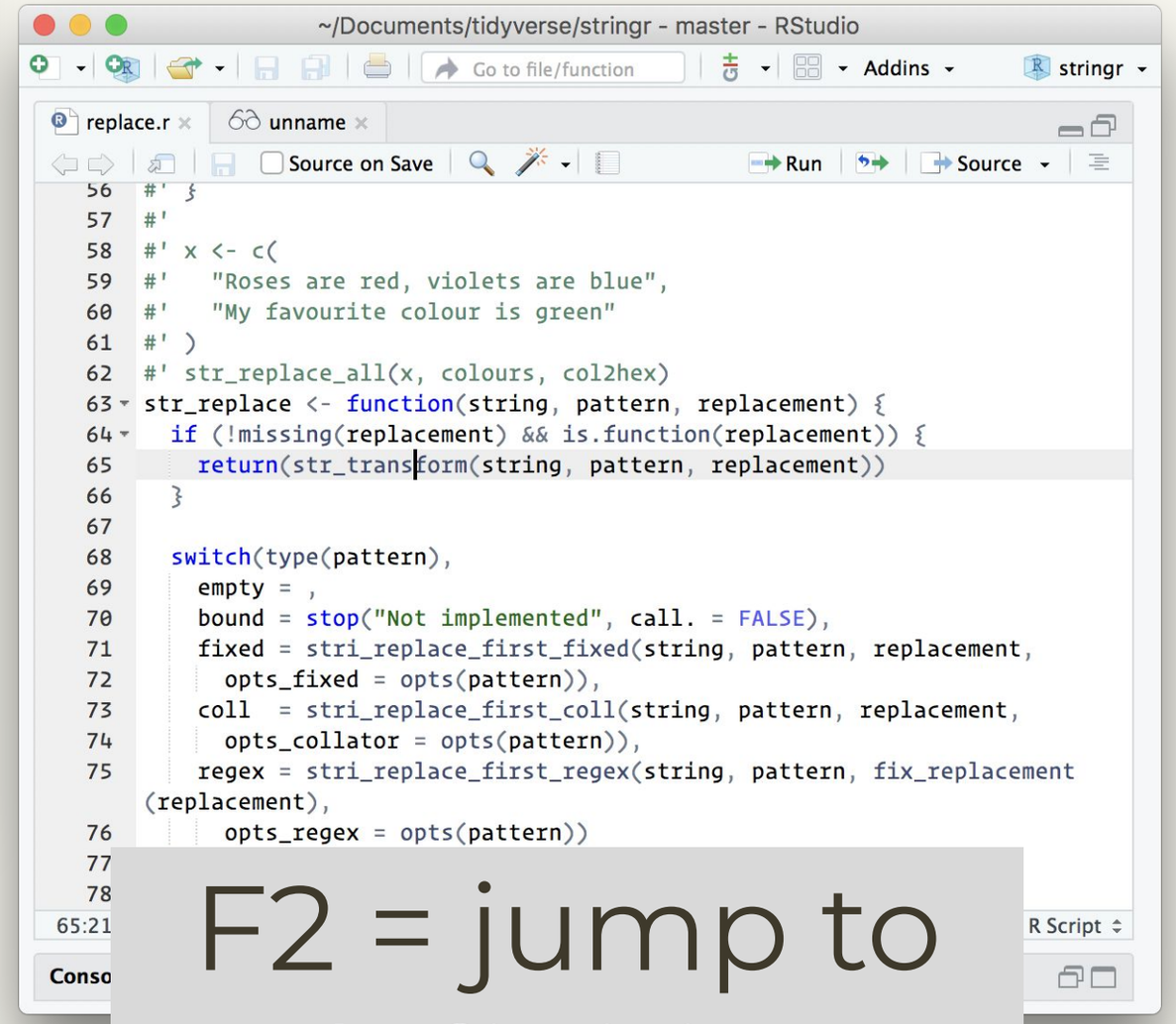
I will come into your lab and SET YOUR COMPUTER ON FIRE 🔥.

— Mash-up of rage tweets by @jennybc and @tpoi.

Enhanced Navigation



Ctrl + . = find functions/files



F2 = jump to definition

Options

General

Code

Appearance

Pane Layout

Packages

Sweave

Spelling

Git/SVN

Publishing

Default working directory (when not in a project):

~ Browse...

☑ Restore most recently opened project at startup

☑ Restore previously open source documents at startup

☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

☑ Always save history (even when not saving .RData)

☑ Remove duplicate entries in history

☐ Use debug error handler only when my code contains errors

☐ Automatically expand tracebacks in error inspector

Default text encoding:

UTF-8 Change...

☑ Automatically notify me of updates to RStudio

OK Cancel Apply

My first package

Your turn

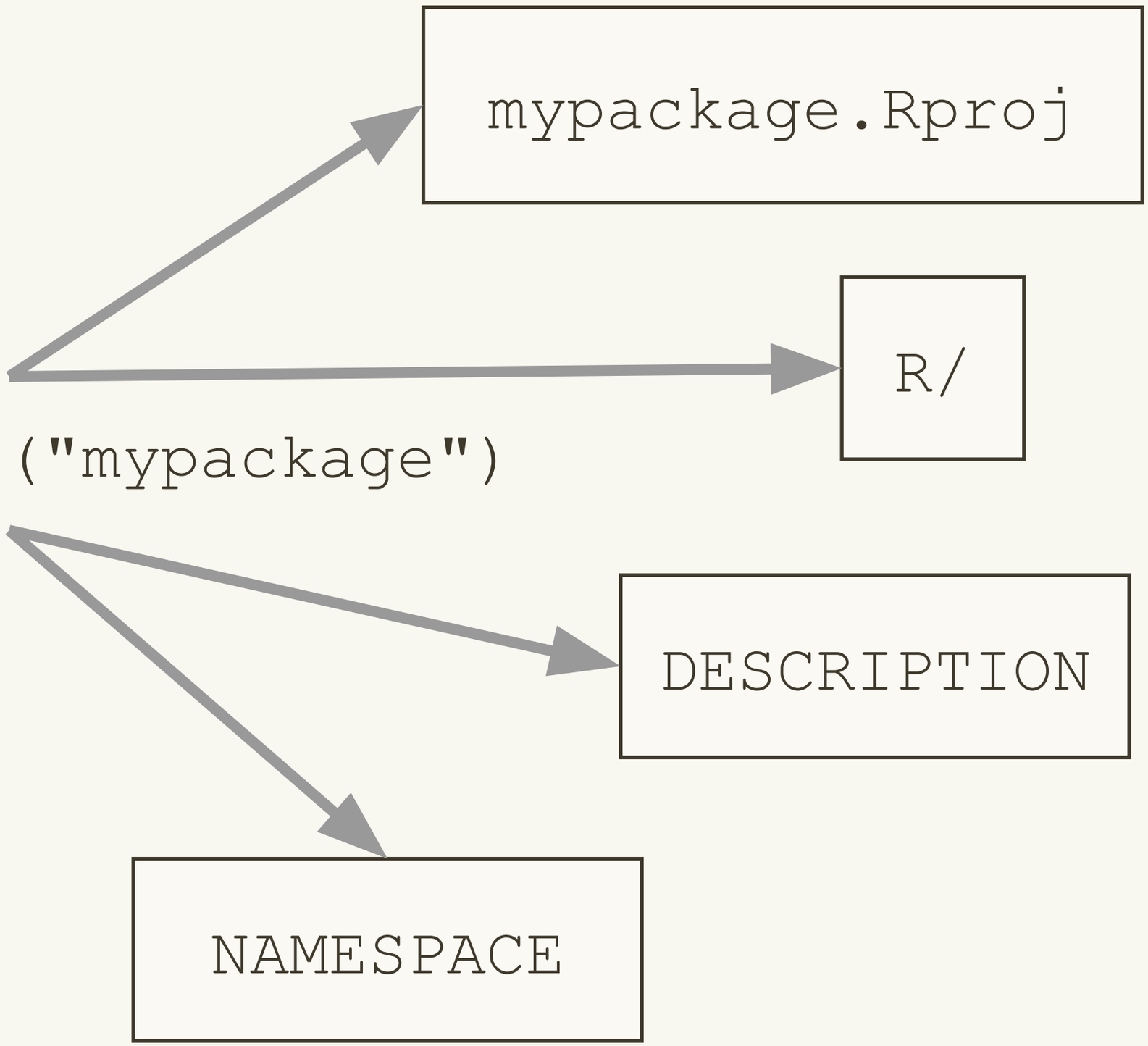
```
# Verify that you can create a package with:
usethis::create_package("~/Desktop/mypackage")

# 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()?

```
create_package("mypackage")
```

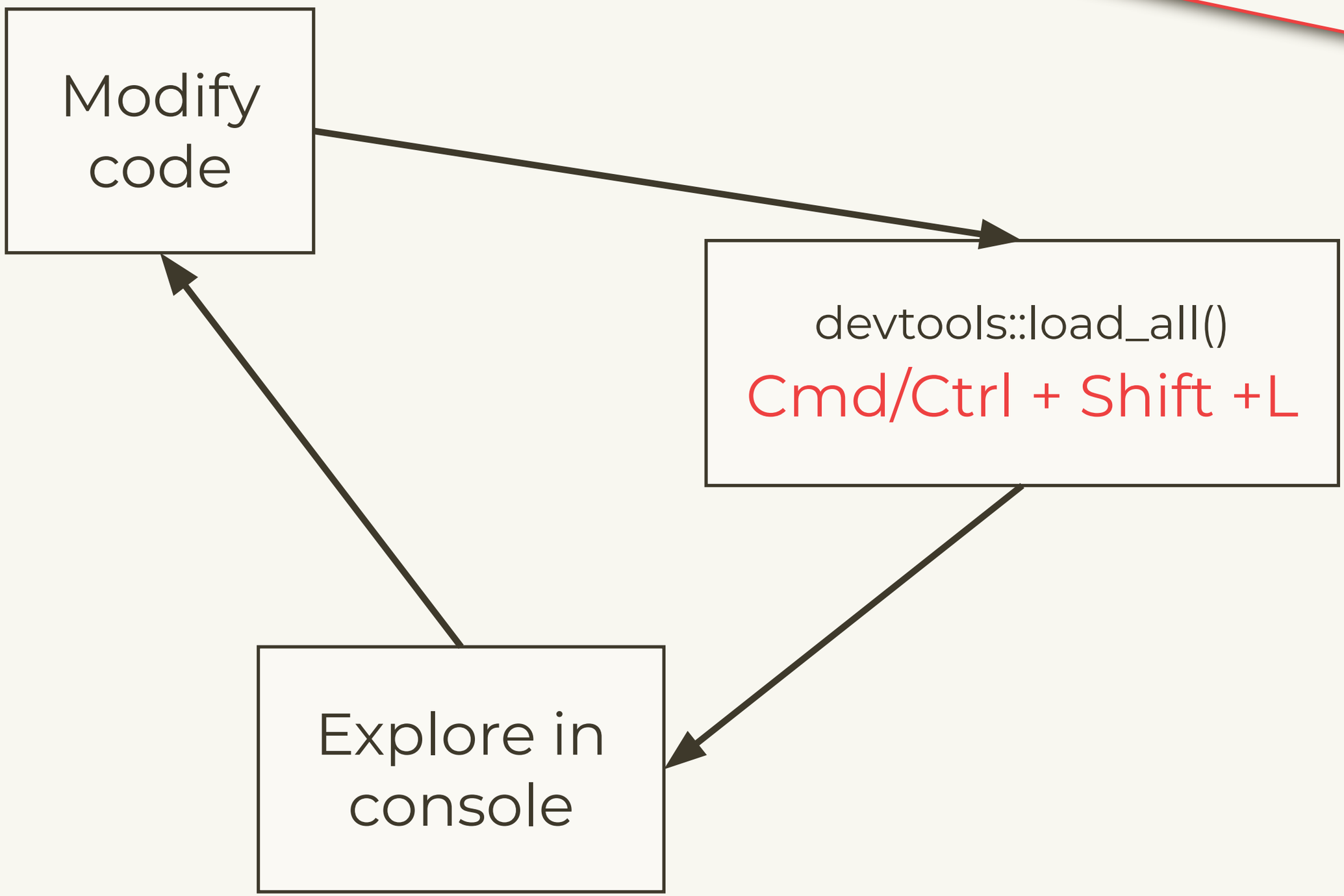


```
package.skeleton()
```

Never use this!

Why bother?

You don't even need to save your code!



Now that you have your package, what do you put in it?

```
# There's a usethis helper adding files!
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

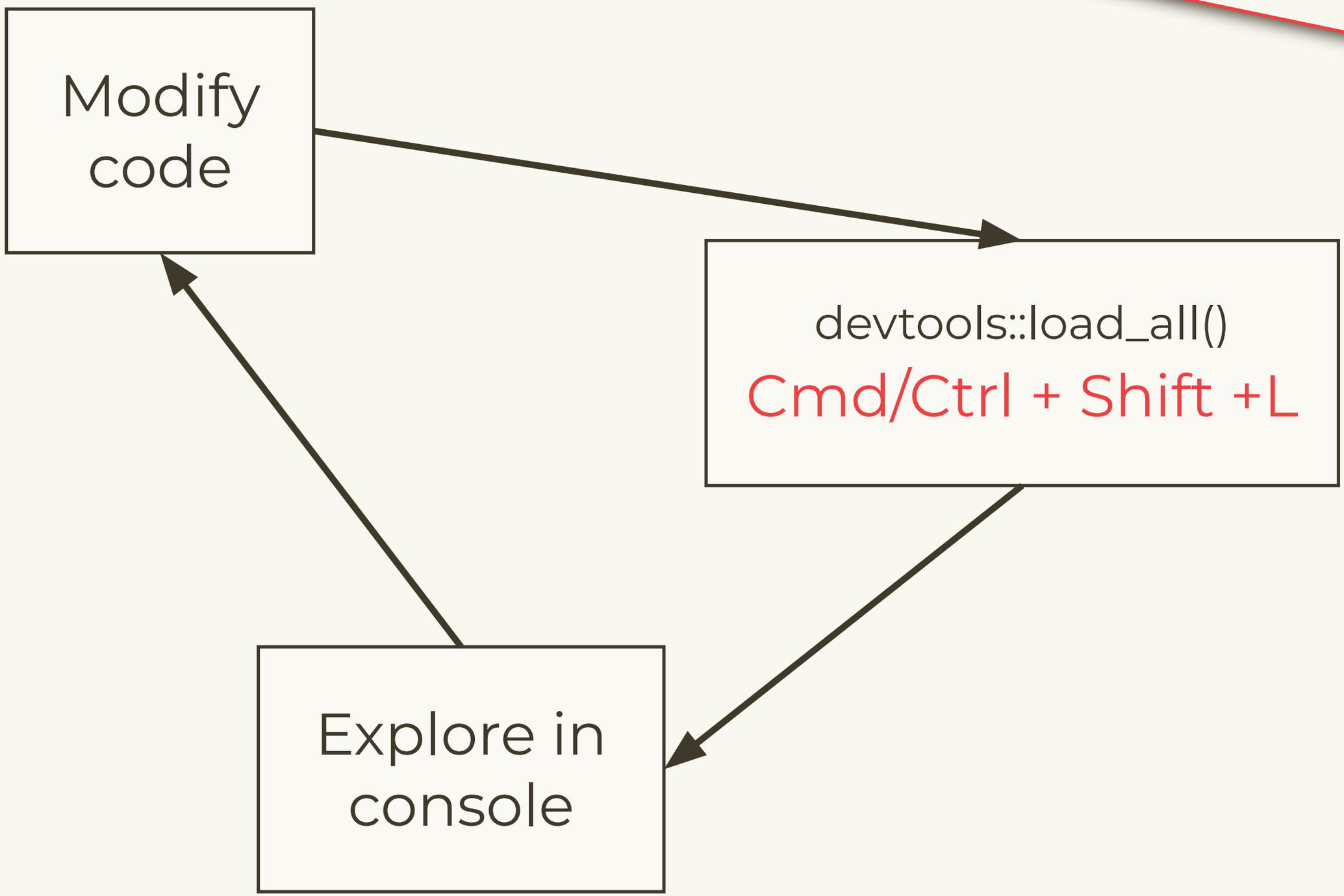
- Create a new R file in your package called "zooSounds.R"
- Paste the following code into your script:

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

- *Hint: a completed version is on github*

Try it out!

You don't even need to save your code!



Your turn

- Change some tiny thing about your function - maybe the animal “says” instead of “goes”?

- Load all with `devtools::load_all()`

- Try adding yourself as an author to the package in DESCRIPTION, and a fun title and description

Woohoo, you did it!

Dependencies

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

What are dependencies?

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)

Use :: to access functions in imported packages

```
# In DESCRIPTION
Imports: foo
```

```
# In bar.R
new_function <- function(x, y, z) {
  foo::bar(x, y) + z
}
```

Should check if suggested package available

```
# In DESCRIPTION
Suggests: foo

# In bar.R
new_function <- function(x, y, z) {
  if (!requireNamespace("foo", quietly =
TRUE)) {
    stop("Need foo! Use
install.packages('foo').")
  }
  foo::bar(x, y) + z
}
```

Reasons to use depends instead of imports

This page has been intentionally left blank

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

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))]
}
```

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!
```

Workflow setup: your .Rprofile

```
# Setup some code that is run every time
# you start R
# usethis::edit_r_profile()

if (interactive()) {
  suppressMessages(require(devtools))
  suppressMessages(require(usethis))
  suppressMessages(require(testthat))
}
```

Never include analysis packages here

```
if (interactive()) {
  suppressMessages(require(ggplot2))
  suppressMessages(require(dplyr))
}
```

While you're in there, also add

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

Your turn

- Follow the instructions in previous slides and make sure that you're optimally configured.

What about scripts?

Currently no strong convention for scripts

It’s optional — you don’t have to put an analysis in a package, and not all packages will have analyses.

For today, I recommend using `analysis/`. There are no tools, and it might change in the future, but it’s what I’m currently thinking

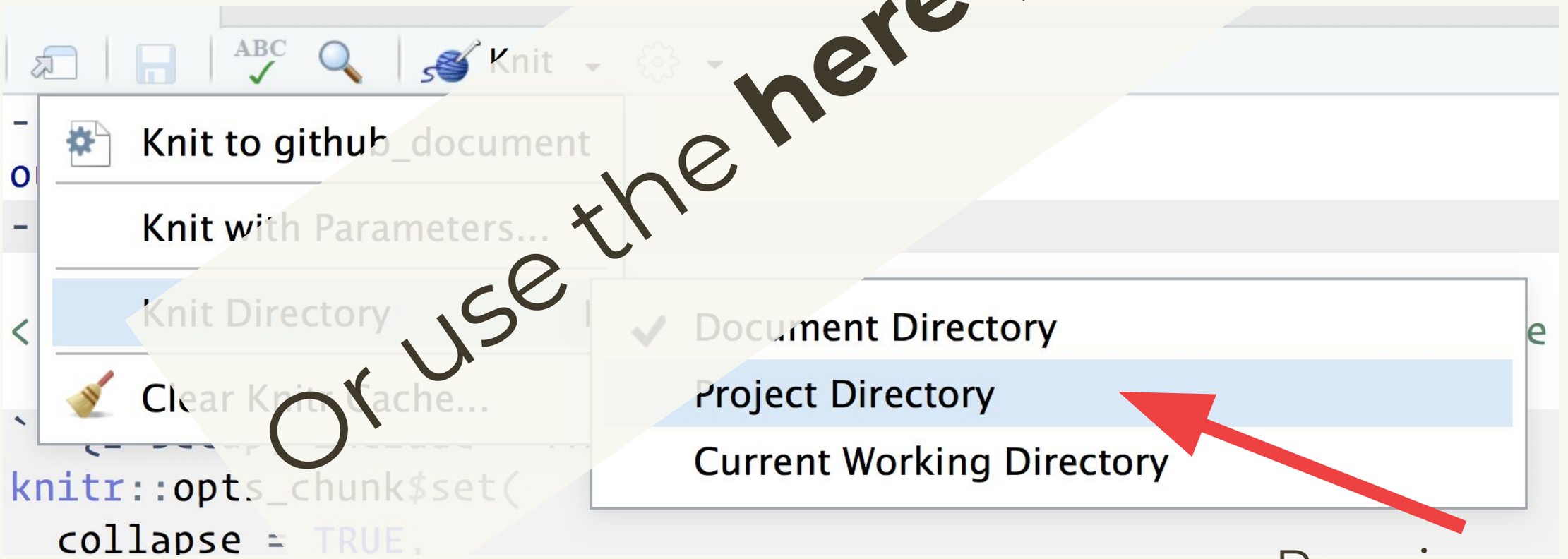
If you want to use the package in the scripts, you’ll need to build and reload so that the package is installed and available.

Beware working directory conventions

When you run R code, the working directory is the project directory

When you knit an Rmd, the working directory is changed to the document directory.

Can change from Knitr menu:



Or use the **here** package

Requires RStudio 1.1