# Writing R Packages

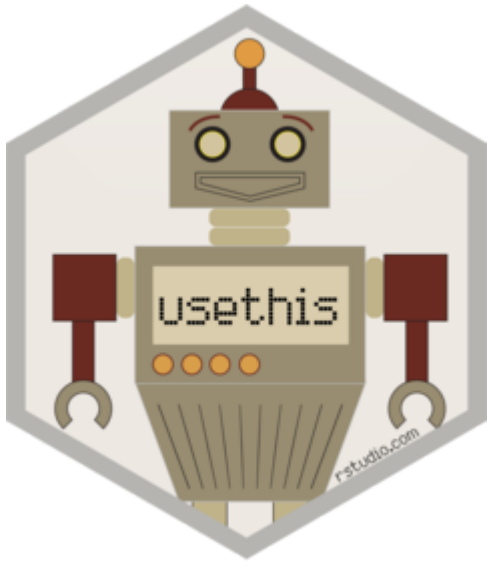
## with Rstudio, {usethis}, and {roxygen2}

Daniel D. Sjoberg

Memorial Sloan Kettering Cancer Center
Department of Epidemiology and Biostatistics

March 28, 2019


Studio®




# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Outline

- *R Package Structure*
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

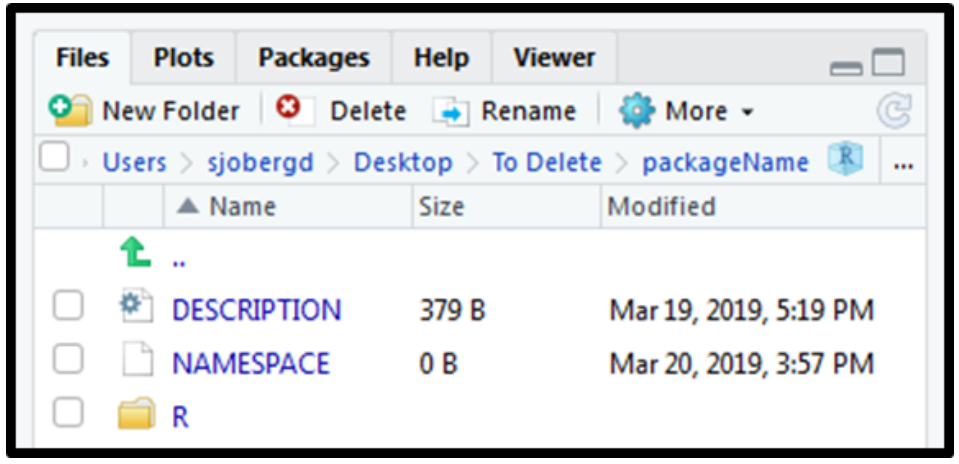
# Package Structure

An R package needs 3 components

1. DESCRIPTION file

2. NAMESPACE file

3. R code folder



# Package Structure

An R package needs 3 components

1. ***DESCRIPTION file***
2. NAMESPACE file
3. R code folder

```
Package: mypackage
Title: What The Package Does (one line)
Version: 0.1
Authors@R: person(
  "First", "Last",
  email = "first.last@example.com",
  role = c("aut", "cre"))
Description: What the package does
  (one paragraph)
Depends: R (>= 3.5)
Imports: dplyr
License: What license is it under?
LazyData: true
```

- store metadata about the package
- list dependencies
- specify version number of package

# Package Structure

An R package needs 3 components

1. DESCRIPTION file

2. ***NAMESPACE file***

3. R code folder

```
# Generated by roxygen2: do not edit by hand

export(tbl_regression)
export(tbl_summary)
export(tbl_uvregression)
importFrom(glue,glue)
importFrom(knitr,knit_print)
importFrom(magrittr,"%>%")
```

- {roxygen2} will take care of this for you!
- lists functions that will be exported by your package
- lists functions imported from other packages

# Package Structure

An R package needs 3 components

1. DESCRIPTION file
2. NAMESPACE file
3. ***R code folder***

- R folder contains R code for each function in your package
  - typically, one code file for each exported function (although not required)
  - e.g. `myfirstfunction.R`
- also contains code for helper or utility functions
  - these functions are not exported, that is, not available to users of the package
  - utility function files begin with utils- prefix
  - e.g. `utils-myfirstfunction.R`

# Package Structure

An R package needs 3 components

1. DESCRIPTION file
2. NAMESPACE file
3. R code folder

The {usethis} package has functions that create the package structure for you

After any function in {usethis} is run, it prints additional information into the console. READ AND FOLLOW THE DIRECTIONS!

- lists files created
- lists files modified
- lists user instructions

{usethis} makes package development a breeze

# Outline

- R Package Structure
- *Getting Started*
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Getting Started

A few functions to get you started with a new package

- `usethis::create_package()`
- `usethis::use_package_doc()`
- `usethis::use_git()`
- `usethis::use_github()`

# Getting Started

A few functions to get you started with a new package

- ***`usethis::create_package()`***
- `usethis::use_package_doc()`
- `usethis::use_git()`
- `usethis::use_github()`

```
> usethis::create_package("~/myPackage")
✓ Setting active project to '~/myPackage'
✓ Creating 'R/'
✓ Creating 'man/'
✓ Writing 'DESCRIPTION'
✓ Writing 'NAMESPACE'
✓ Writing 'myPackage.Rproj'
✓ Adding '.Rproj.user' to '.gitignore'
✓ Adding '^myPackage\\.Rproj$', '^\\.Rproj\\.user$'
   to '.Rbuildignore'
✓ Opening new project 'myPackage' in RStudio
```

- Create package folder and a skeleton of the folder structure

# Getting Started

A few functions to get you started with a new package

- `usethis::create_package()`
- ***`usethis::use_package_doc()`***
- `usethis::use_git()`
- `usethis::use_github()`

```
> usethis::use_package_doc()
✓ Writing 'R/myPackage-package.R'
```

'R/myPackage-package.R' contents

```
#' @keywords internal
"_PACKAGE"
```

- writes a basic documentation file for you package
- we will add more to this later

# Getting Started

A few functions to get you started with a new package

- `usethis::create_package()`
- `usethis::use_package_doc()`
- ***`usethis::use_git()`***
- `usethis::use_github()`

```
> usethis::use_git()
✓ Initialising Git repo
✓ Adding '.Rhistory', '.RData' to '.gitignore'
OK to make an initial commit of 6 files?
1: Negative
2: Not now
3: Yeah

Selection: 3
✓ Adding files and committing
```

- create a git repository
- commit existing files to the repo

# Getting Started

A few functions to get you started with a new package

- `usethis::create_package()`
- `usethis::use_package_doc()`
- `usethis::use_git()`
- ***`usethis::use_github()`***

```
> usethis::use_github()
● Check title and description
  Name:        myPackage
  Description: What the Package Does (One Line)
Are title and description ok?
1: No
2: Nope
3: Yeah

Selection: 3
✓ Creating GitHub repository
✓ Adding GitHub remote
✓ Adding GitHub links to DESCRIPTION
✓ Setting URL field in DESCRIPTION to
  'https://github.com/ddsjoberg/myPackage'
✓ Setting BugReports field in DESCRIPTION to
  'https://github.com/ddsjoberg/myPackage/issues'
✓ Pushing to GitHub and setting
  remote tracking branch
```

# Getting Started

A few functions to get you started with a new package

- `usethis::create_package()`
- `usethis::use_package_doc()`
- `usethis::use_git()`
- ***`usethis::use_github()`***

- THIS ONLY WORKS IF YOU'VE PREVIOUSLY CONFIGURED THE `use_github()` FUNCTION
- recommend you setup Rstudio to play nicely with GitHub. Read *Happy Git and GitHub for the useR* for details (https://happygitwithr.com/)
- you can create your GitHub repo manually and add the package contents if you haven't yet configured RStudio and GitHub
    - remember to update the url and bug reports url in the DESCRIPTION file to match the GitHub repo location

# Outline

- R Package Structure
- Getting Started
- ***Adding Functions***
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Adding Functions

```
usethis::use_r()
```

- creates a new code file for you write your function
- places file correctly in the `R` folder
- the new file is entirely blank

```
> usethis::use_r("my_mean")
● Modify 'R/my_mean.R'
```

# Adding Functions

Let's write our first function

```
my_mean <- function(x) {
  mean(na.omit(x))
}
```

For EVERY non-base R function you need to either *import* the function, or use `::` to reference the function

```
my_mean <- function(x) {
  mean(stats::na.omit(x))
}
```

We will now use {roxygen2} comments in our code to document our new function (aka write the help file)!

# Adding Functions: Documentation

- R function help files (*.Rd) are saved in the *man* folder
- the *man* folder already exists courtesy of `usethis::create_package()`
- R processes the *.Rd files to create plain text, PDF, and HTML versions of the help files
- the code in *.Rd files looks somewhat like LaTeX: it's verbose and cumbersome to write
- we will automate the creation of these files with {roxygen2} comments
- by automating, we link the function code to the documentation
- this helps keep the documentation up to date

```
> usethis::use_roxygen_md()
✓ Setting Roxygen field in DESCRIPTION to 'list(markdown = TRUE)'
✓ Setting RoxygenNote field in DESCRIPTION to '6.1.1'
● Run `devtools::document()`
```

# Adding Functions: Documentation

- roxygen comments appear above a function
- roxygen comment lines always begins with `#'`
- two common roxygen tags
  - `@param` used to document a function argument
  - `@export` tells roxygen to export the function when the package is built

```
#' The first line is the title
#'
#' The second section is a longer description of the function.
#' This can go on for multiple lines.
#'
#' @param x numeric vector
#' @export
my_mean <- function(x) {
  mean(stats::na.omit(x))
}
```

# Adding Functions: Documentation

- other notable roxygen tags
  - `@seealso` list related references (typically used to reference related functions)
  - `@family` similar to `@seealso`, but creates a list of "see also" functions that belong to the same family
  - `@examples` add examples to function help file
  - `@author` list author(s) of the function
  - `@return` specify the returned object
- link to other functions in help file
  - function in the same package `\code{\link{my_mean}}`
  - function in another package `\code{\link[base]{mean}}`
- Blog post with more details
  - http://kbroman.org/pkg_primer/pages/docs.html

# Adding Functions: Documentation

```r
#' The first line is the title
#'
#' The second section is a longer
#' description of the function.
#' This can go on for multiple lines.
#'
#' @param x numeric vector
#' @export
#' @seealso \code{\link[base]{mean}}
#' @author Daniel D. Sjoberg
#' @examples
#' my_mean(1:5)
my_mean <- function(x) {
  mean(stats::na.omit(x))
}
```

my_mean {myPackage} R Documentation

## The first line is the title

### Description

The second section is a longer description of the function. This can go on for multiple lines.

### Usage

```
my_mean(x)
```

### Arguments

| | |
|---|---|
| x | numeric vector |

### Author(s)

Daniel D. Sjoberg

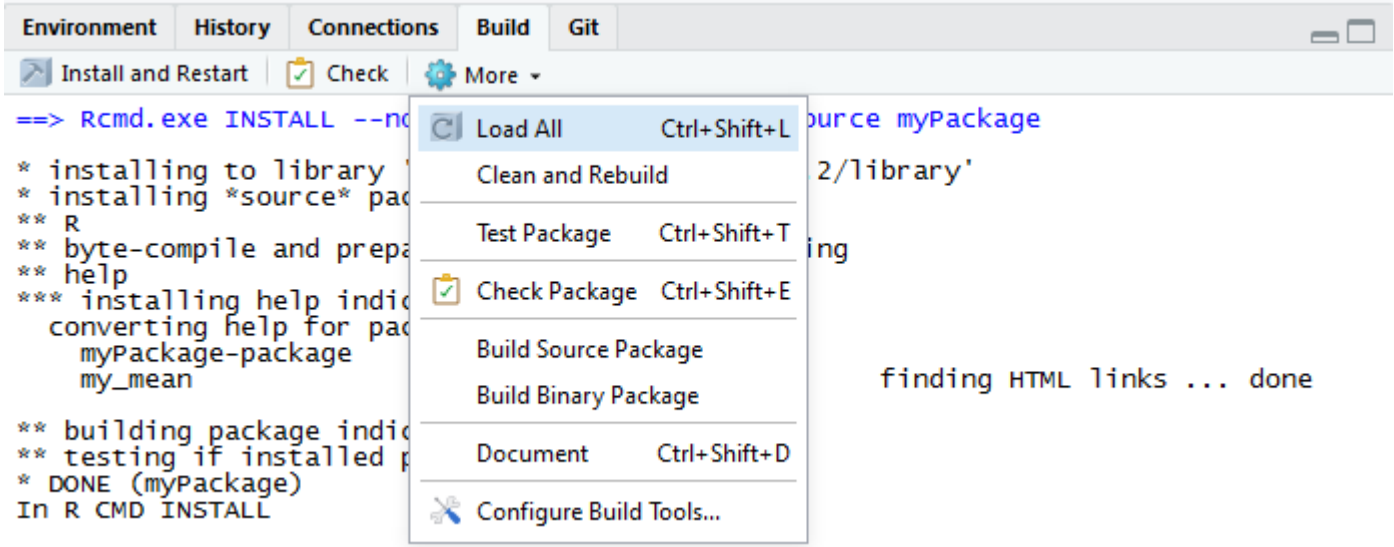
### See Also

mean

### Examples

```
my_mean(1:5)
```

# Adding Functions: Rstudio Build Tab




**Document** each time you update roxygen comments

# Adding Functions

- new function calculates mean of every column in a data frame
- use the `map()` function in the {purrr} package
- use `::` to refer to the function: `purrr::map()`
- document that our package now depends on {purrr}
- `usethis::use_package("purrr")`

```
> usethis::use_package("purrr")
✓ Setting active project to '~/myPackage'
✓ Adding 'purrr' to Imports field
  in DESCRIPTION
● Refer to functions with `purrr::fun()`
```

`DESCRIPTION` (truncated)

```
Imports:
    purrr
```

---

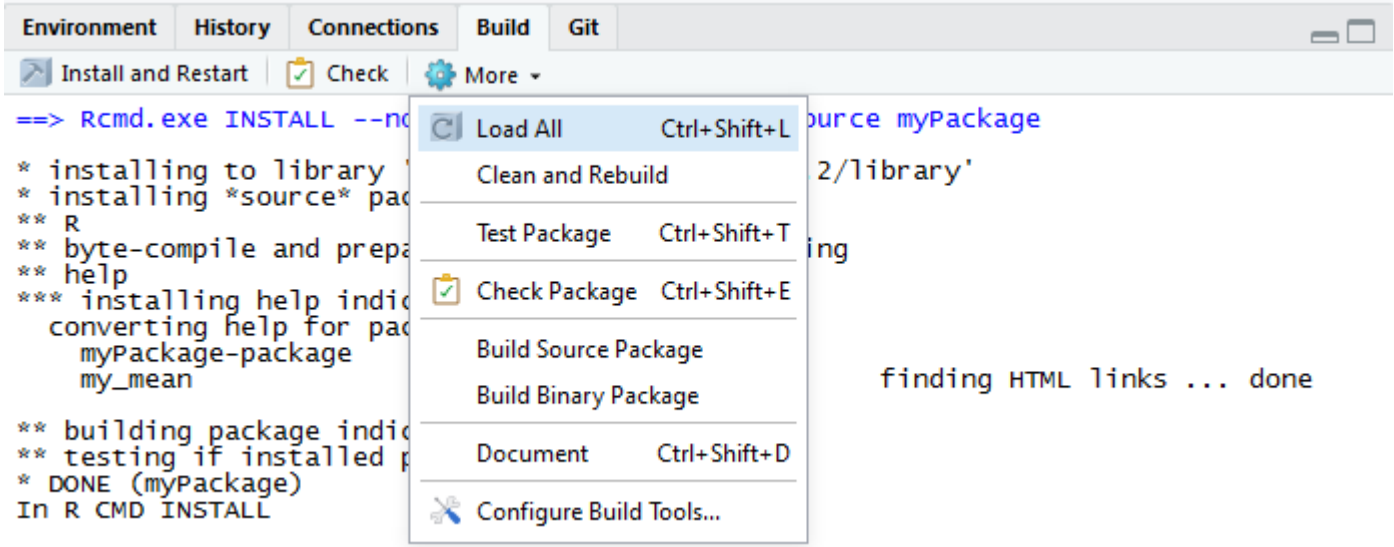
```
df_mean <- function(data) {
  purrr::map(data, my_mean)
}
```

```
> df_mean(mtcars)
$mpg
[1] 20.09062

$cyl
[1] 6.1875

$disp
[1] 230.7219
```

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- ***Build your Package***
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Build your Package: Rstudio Build Tab




- **Document** each time you update roxygen comments
- **Install and Restart** each time you update R code
- **Check**

# Build your Package

What is checked?
So much! Here's a very abbreviated list

- package structure
  - hidden files/folders
  - portable file names
  - executable files
  - package subdirectories
  - left-over files
- DESCRIPTION/NAMESPACE file
  - package dependencies
  - files exist
  - NAMESPACE parses properly
- R code
  - non-ASCII characters
  - syntax errors
  - dependencies in R code
  - S3 generic/method consistency
- documentation
  - Rd/help files
  - Rd file metadata
  - examples
  - undocumented function arguments

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- ***Unit Tests***
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Unit Tests

- vital part of package development
- helps ensure future updates don't break functioning code
- easily implemented with the {testthat} package
- all unit tests are run each time the package is checked

```
> usethis::use_testthat()
✓ Setting active project to '~/myPackage'
✓ Adding 'testthat' to
  Suggests field in DESCRIPTION
✓ Creating 'tests/testthat/'
✓ Writing 'tests/testthat.R'

> usethis::use_test("df_mean")
✓ Writing 'tests/testthat/test-df_mean.R'
● Modify 'tests/testthat/test-df_mean.R'
```

**tests/testthat/test-df_mean.R**

```
context("test-df_mean")

test_that("multiplication works", {
  expect_equal(2 * 2, 4)
})
```

# Unit Tests

There are many types of checks that can be included

- `expect_lte()`
- `expect_gte()`
- `expect_equal()`
- `expect_setequal()`
- `expect_equivalent()`
- `expect_identical()`
- `expect_length()`
- `expect_null()`
- `expect_error()`
- `expect_warning()`
- `expect_true()`

Each test should

1. have an informative name
2. cover a single unit of functionality

The idea is that when a test fails, you'll know what's wrong and where in your code to look for the problem.

Hadley Wickham's book on writing R packages provides a thorough review on writing unit tests.

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

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- ***Advanced Setup***
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Advanced Setup

- *software license*
- `usethis::use_news_md()`
- `usethis::use_spell_check()`
- continuous integration
- `usethis::use_coverage()`

Specify the license you want associated with your package.

- `usethis::use_mit_license()`
- `usethis::use_gpl3_license()`
- `usethis::use_lgpl_license()`
- `usethis::use_apl2_license()`
- `usethis::use_cc0_license()`

```
> usethis::use_mit_license("Daniel D. Sjoberg")
✓ Setting License field in DESCRIPTION
     to 'MIT + file LICENSE'
✓ Writing 'LICENSE.md'
✓ Adding '^LICENSE\\.md$' to '.Rbuildignore'
✓ Writing 'LICENSE'
```

# Advanced Setup

- software license
- ***`usethis::use_news_md()`***
- `usethis::use_spell_check()`
- continuous integration
- `usethis::use_coverage()`

  - creates a basic `NEWS.md` in the root directory
  - keep track of updates and versions
  - communicate changes to API to users (also future you)

```
> usethis::use_news_md()
✓ Writing 'NEWS.md'
● Modify 'NEWS.md'
```

# Advanced Setup

- software license
- `usethis::use_news_md()`
- ***`usethis::use_spell_check()`***
- continuous integration
- `usethis::use_coverage()`

- adds a unit test to automatically run a spell check on documentation and vignettes
- adds a WORDLIST file to the package, which is a dictionary of white-listed words
- runs when the package checks are run
- suggest including `error = TRUE` option so spelling errors fail package checks

```
> usethis::use_spell_check(error = TRUE)
Updated ~/myPackage/tests/spelling.R
● Run `devtools::check()` to trigger spell check
```

# Advanced Setup

- software license
- `usethis::use_news_md()`
- `usethis::use_spell_check()`
- ***continuous integration***
- `usethis::use_coverage()`


- get cool badges
- others more confident in your package
- `usethis::use_travis()` tests build on Linux: http://travis-ci.org
- `usethis::use_appveyor()` tests build on Windows: https://www.appveyor.com/
- tests are triggered each time package is pushed to GitHub
- blog post about using CI https://juliasilge.com/blog/beginners-guide-to-travis/
- requires public GitHub repo

# Advanced Setup

- software license
- `usethis::use_news_md()`
- `usethis::use_spell_check()`
- ***continuous integration***
- `usethis::use_coverage()`

Be sure to follow all directions!

```
> usethis::use_travis()
✓ Writing '.travis.yml'
✓ Adding '^\\.travis\\.yml$' to '.Rbuildignore'
● Turn on travis for your repo at
    https://travis-ci.org/profile/ddsjoberg
● Add a Travis build status badge by adding
    the following line to your README:
Copying code to clipboard:
  [![Travis build status](https://travis-ci.org/ddsjo

> usethis::use_appveyor()
✓ Writing 'appveyor.yml'
✓ Adding '^appveyor\\.yml$' to '.Rbuildignore'
● Turn on AppVeyor for this repo at
    https://ci.appveyor.com/projects/new
● Add a AppVeyor build status badge by adding
    the following line to your README:
Copying code to clipboard:
  [![AppVeyor build status](https://ci.appveyor.com/a
```

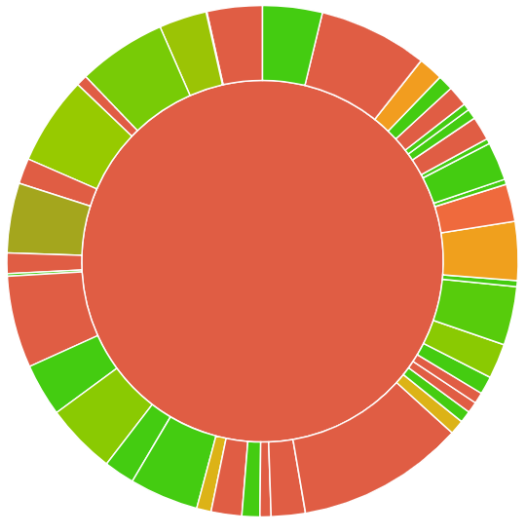
# Advanced Setup

- software license
- `usethis::use_news_md()`
- `usethis::use_spell_check()`
- continuous integration
- ***`usethis::use_coverage()`***

- more cool badges
- adds test coverage reports to a package that is already using Travis CI
- reports the proportion of code covered by unit tests
- provides reports of coverage of every file, line-by-line
- https://codecov.io/

# Advanced Setup

- software license
- `usethis::use_news_md()`
- `usethis::use_spell_check()`
- continuous integration
- ***`usethis::use_coverage()`***



/ R / utils-table1_input_checks.R

{usethis} prints directions into the console...be sure to follow them!

```
> usethis::use_coverage()
✓ Adding 'covr' to Suggests field in DESCRIPTION
✓ Writing 'codecov.yml'
✓ Adding '^codecov\.yml$' to '.Rbuildignore'
● Add a Coverage status badge by adding the
    following line to your README:
Copying code to clipboard:
  [![Coverage status](https://codecov.io/gh/ddsjoberg
● Add to '.travis.yml':
Copying code to clipboard:
  after_success:
    - Rscript -e 'covr::codecov()'
```

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- *Advanced Documentation*
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Advanced Documentation: README

- GitHub renders the README file and is often the first thing users will see about your package
- rather than use a basic README.md, use an R markdown README.rmd
- `usethis::use_readme_rmd()`

```
> usethis::use_readme_rmd()
✓ Writing 'README.Rmd'
✓ Adding '^README\\.Rmd$' to '.Rbuildignore'
● Modify 'README.Rmd'
✓ Writing '.git/hooks/pre-commit'
```

- update and knit README.rmd, and it'll create README.md
  - README.md now has this header: `<!-- README.md is generated from README.Rmd. Please edit that file -->`
- be sure to knit README.rmd after you modify the file OR update functions highlighted in the README file

# Advanced Documentation: Vignettes

- a vignette is a long-form guide to your package
- describes the problem your package is designed to solve, then shows the reader how to solve it

```
> usethis::use_vignette("my first vignette")
#> ✓ Adding 'knitr' to Suggests field in DESCRIPTION
#> ✓ Setting VignetteBuilder field in
#>     DESCRIPTION to 'knitr'
#> ✓ Adding 'rmarkdown' to Suggests field
#>     in DESCRIPTION
#> ✓ Creating 'vignettes/'
#> ✓ Adding '*.html', '*.R' to 'vignettes/.gitignore'
#> ✓ Adding 'inst/doc' to '.gitignore'
#> ✓ Creating 'vignettes/my-first-vignette.Rmd'
#> ● Modify 'vignettes/my-first-vignette.Rmd'
```

- a vignette is an R markdown document with HTML output
- be sure to change the title in BOTH `title:` and `VignetteIndexEntry{}` in the vignette yaml

```
title: "Vignette Title"
author: "Vignette Author"
date: "2019-04-01"
output: rmarkdown::html_vignette
vignette: >
%\VignetteIndexEntry{Vignette Title}
%\VignetteEngine{knitr::rmarkdown}
\usepackage[utf8]{inputenc}
```

- R packages, Vignettes Chapter: http://r-pkgs.had.co.nz/vignettes.html

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- ***Include Datasets***
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Include Datasets

- including datasets in an R package is easy with `use_data_raw()` and `use_data()`
- we'll go over an example where we simulate a dataset and save it in the package

```
> usethis::use_data_raw()
#> ✓ Creating 'data-raw/'
#> ✓ Adding '^data-raw$' to '.Rbuildignore'
#> Next:
#> • Add data creation scripts
#>     in 'data-raw/'
#> • Use `usethis::use_data()` to
#>     add data to package
```

- simulate dataset, and `use_data()` will save it with the package
- save this script is the `data-raw/` folder

```
set.seed(8976) # remeber to set your seed!
my_data <-
  tibble::tibble(
    x = runif(100),
    y = runif(100)
  )

usethis::use_data(my_data, overwrite = TRUE)
#> ✓ Creating 'data/'
#> ✓ Saving 'my_data' to 'data/my_data.rda'
```

# Include Datasets

- document the dataset
- use {roxygen2} comments for documentation
- add a file to the R folder to begin documenting
- I call mine `data.R`

```
> usethis::use_r("data")
● Modify 'R/data.R'
```

`R/data.R`

```
#' My simulated data
#'
#' A simulated dataset that I am saving in my package
#'
#' @format A data frame with 100 rows
#' \describe{
#'     \item{x}{Random Uniform Variable}
#'     \item{y}{Random Uniform Variable}
#' }
"my_data"
```

- after you write the {roxygen2} comments, document the package
- the help file will be accessible with `?my_data` after the package has been loaded

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- ***Other Tips***
- Non-Standard Evaluation
- Collaborate with GitHub
- Create Package Website

# Other Tips

- {roxygen2} tags `@import` and `@importFrom`
- `use_pipe()` to import and export {magrittr} pipe operator
- helper functions for tidy development

# Other Tips: Import

- there are a few functions from the tidyverse we use frequently
- referring to them with `::` quickly becomes cumbersome
- we can import functions and even entire packages to avoid the `::` notation
- {roxygen2} tags `@import` and `@importFrom`

# Other Tips: Import

- remember that boring package-level documentation file we made with `use_package_doc()`? let's use this file to import commonly used functions

```
#' @import dplyr
#' @import purrr
#' @importFrom tidyr nest unnest
     spread gather complete
#' @importFrom tibble tibble as_tibble
#' @importFrom rlang .data %||% set_names sym
     expr enexpr quo enquo parse_expr
#' @importFrom glue glue
#' @keywords internal
"_PACKAGE"

# allowing for the use of the dot when piping
utils::globalVariables(".")
```

- when adding entire package imports, be sure to run the package checks. you may experience warnings from conflicting function names
    - this is common with tidyverse packages as many of them contain the same functions
- `@import` and `@importFrom` can be placed in any R code file
    - can import same function in multiple files. {roxygen2} will sort out duplicates when the package is documented
- use the package-level documentation file for imports that apply to all functions in your package

# Other Tips: the pipe

- the {magrittr} pipe operator is so useful, you may want to both import and export it to make it available to users of your package

```
> usethis::use_pipe()
✓ Adding 'magrittr' to Imports
    field in DESCRIPTION
✓ Writing 'R/utils-pipe.R'
● Run `devtools::document()`
```

R/utils-pipe.R

```
#' Pipe operator
#'
#' See \code{magrittr::\link[magrittr]{\%>\%}}
#' for details.
#'
#' @name %>%
#' @rdname pipe
#' @keywords internal
#' @export
#' @importFrom magrittr %>%
#' @usage lhs \%>\% rhs
NULL
```

# Other Tips: tidy helpers

the {usethis} package contains helper functions to develop tidy packages

these are my favorites

- `use_tidy_description()`
  - puts fields in standard order and alphabetizes dependencies in DESCRIPTION file
- `use_tidy_versions(overwrite = FALSE)`
  - pins all dependencies to require at least the currently installed version
  - helps ensure your package will work on all systems
- `use_tidy_style(strict = TRUE)`
  - uses the {styler} package to style all code according to the tidyverse style guide
  - keeps your code easy to read, looking good, and collaborative

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- ***Non-Standard Evaluation***
- Collaborate with GitHub
- Create Package Website

# Non-Standard Evaluation

- tidyverse code is quick to write, but as a result it is ambiguous
  - these equivalent `mutate` statements return different results
  - the code in your package must be precise

```
y = 10
tibble(
  x = 1:4
) %>%
  mutate(
    xy = x + y
  )
```

```
## # A tibble: 4 x 2
##       x    xy
##   <int> <dbl>
## 1     1    11
## 2     2    12
## 3     3    13
## 4     4    14
```

```
tibble(
  y = 4:1,
  x = 1:4
) %>%
  mutate(
    xy = x + y
  )
```

```
## # A tibble: 4 x 3
##       y     x    xy
##   <int> <int> <int>
## 1     4     1     5
## 2     3     2     5
## 3     2     3     5
## 4     1     4     5
```

# Non-Standard Evaluation

- how to make the code precise?
  - import the `.data` function from {rlang}
  - use `.data$<varname>` EVERY time you reference a column within a `mutate` statement
  - if you don't use `.data`, the package check will return a warning

```
y = 10
tibble(
  x = 1:4
) %>%
  mutate(
    xy = .data$x + y
  )
```

```
## # A tibble: 4 x 2
##       x    xy
##   <int> <dbl>
## 1     1    11
## 2     2    12
## 3     3    13
## 4     4    14
```

```
tibble(
  y = 4:1,
  x = 1:4
) %>%
  mutate(
    xy = .data$x + .data$y
  )
```

```
## # A tibble: 4 x 3
##       y     x    xy
##   <int> <int> <int>
## 1     4     1     5
## 2     3     2     5
## 3     2     3     5
## 4     1     4     5
```

# Non-Standard Evaluation

- other tidyverse functions take bare column names as inputs
  - e.g. `mtcars %>% select(mpg, hp)`
  - the reference to the `mpg` and `hp` columns is not standard, and the package check will return warnings if this code is found
- many tidyverse functions will allow you to simply replace the bare references with quoted strings

```
mtcars %>% select(mpg, hp)

mtcars %>% pull(mpg)

mtcars %>% group_by(cyl) %>%
  nest(mpg, hp)
```

```
mtcars %>% select(c("mpg", "hp"))

mtcars %>% pull("mpg")

mtcars %>% group_by(cyl) %>%
  nest(c("mpg", "hp"))
```

# Non-Standard Evaluation

- for functions that do not allow you to simply add a string, use `!!` and `rlang::sym()`
- when you have multiple variables, use `!!!` and `rlang::syms()` instead of `!!` and `rlang::sym()`

```
mtcars %>% group_by(cyl)


mtcars %>% group_by(cyl, am)
```

```
mtcars %>% group_by(!!sym("cyl"))
mtcars %>% group_by(.data$cyl)

by_vars <- c("cyl", "am")
mtcars %>% group_by(!!!syms(by_vars))
mtcars %>% group_by(.data$cyl, .data$am)
```

# Non-Standard Evaluation

- entire expressions can be stored as string and evaluated
- this is particularly powerful when combined with the {glue} package

```
mtcars %>% filter(cyl == 4)
```

```
mtcars %>% filter(!!parse_expr("cyl == 4"))
mtcars %>% filter(.data$cyl == 4)
```

- can also create functions with NSE inputs
- briefly, you replace `sym()` with `enquo()`, more on that in the additional reading
- additional resources
  - *Programming with dplyr*
    https://cran.r-project.org/web/packages/dplyr/vignettes/programming.html
  - *Tidy evaluation, most common actions*
    https://edwinth.github.io/blog/dplyr-recipes/
    - this resource is excellent!

# Outline

- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- ***Collaborate with GitHub***
- Create Package Website

# Collaborate with GitHub

- create a development branch (e.g. *dev*) and do 100% of your initial package building directly on this branch
- once package is standing on its own legs (not necessarily done), make all updates via forks or feature branches
- protect your *master* and *dev* branches
  - from the GitHub repo, select *Settings*, select *Branches*, and add rules for each branch
  - for the *dev* and *master* branches, I recommend
    - "Require pull request reviews before merging"
    - "Require status checks to pass before merging"
    - "Include administrators"

# Outline

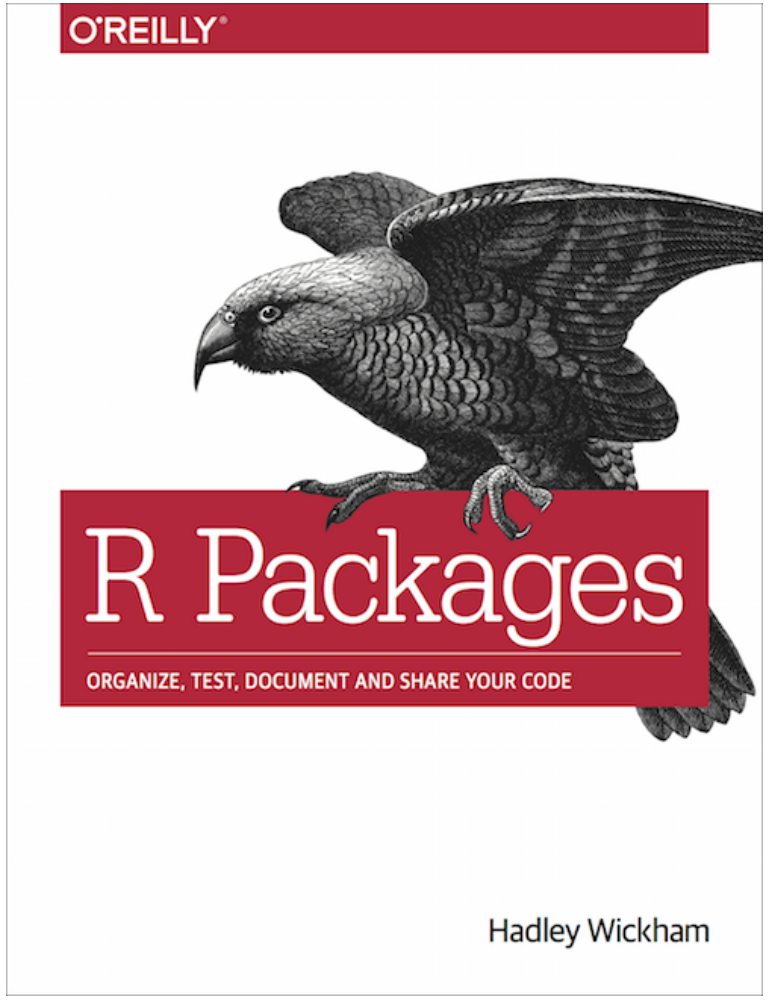
- R Package Structure
- Getting Started
- Adding Functions
- Build your Package
- Unit Tests
- Advanced Setup
- Advanced Documentation
- Include Datasets
- Other Tips
- Non-Standard Evaluation
- Collaborate with GitHub
- ***Create Package Website***

# Create Package Website

- lastly, it's easy to create a website for your package
- easier for a user to navigate, find functions, and
- from the GitHub repo, select *Settings*, scroll down to *GitHub Pages*, from the *Source* menu select *master branch/docs folder*
- the location of the published site is listed under *GitHub Pages*
- run `pkgdown::build_site()`
- done! can take a few minutes to be available online
- more details at https://pkgdown.r-lib.org/

# Resources

- {roxygen2}
  - http://kbroman.org/pkg_primer/pages/docs.html
  - https://blog.rstudio.com/2017/02/01/roxygen2-6-0-0/
- continuous integration
  - https://juliasilge.com/blog/beginners-guide-to-travis/
- *R Packages*, by Hadley Wickham
  - touches on most of the topics covered here (package structures, unit tests, documentation)
  - does not include {usethis} setup
  - http://r-pkgs.had.co.nz/
- Git and GitHub
  - https://happygitwithr.com/



# Thank you

slides at danieldsjoberg.com/writing-R-packages

source code and sample package code at github.com/ddsjoberg/writing-R-packages