How to Share Your Research by Developing R Packages

Stefan Mayer

School of Business and Economics, University of Tübingen

82nd Annual Meeting of the VHB
Frankfurt am Main, March 2020

Materials available at: `https://github.com/stm/vhb_2020`

Motivation

Why write R packages?

Why write an R package (in general)?

- R users already know how to use / install packages
- easy to distribute
- but ...

Why write R packages?

Why write an R package (in general)?

- R users already know how to use / install packages
- easy to distribute
- but ... can also be just for *you*

  → it does not have to go on CRAN / Bioconductor

Why write R packages?

Why write an R package (in general)?

- R users already know how to use / install packages
- easy to distribute
- but ... can also be just for *you*

  → it does not have to go on CRAN / Bioconductor

Why write an R package (as an academic)?

- R packages are a wonderful way to make functions and datasets easily accessible for everyone
  - facilitates that others (academics and practitioners) can *actually* use your research
- actively contribute to open science
- increase the likelihood of citations!

Why write R packages?

Why write an R package (in general)?

- R users already know how to use / install packages
- easy to distribute
- but ... can also be just for *you*

  → it does not have to go on CRAN / Bioconductor

Why write an R package (as an academic)?

- R packages are a wonderful way to make functions and datasets easily accessible for everyone
  - facilitates that others (academics and practitioners) can *actually* use your research
- actively contribute to open science
- increase the likelihood of citations!

Note: Use **zenodo** to share (and preserve) any research with a DOI (so that others can cite you). See for example the R packages *CorporaCoCo* or *foieGras*.

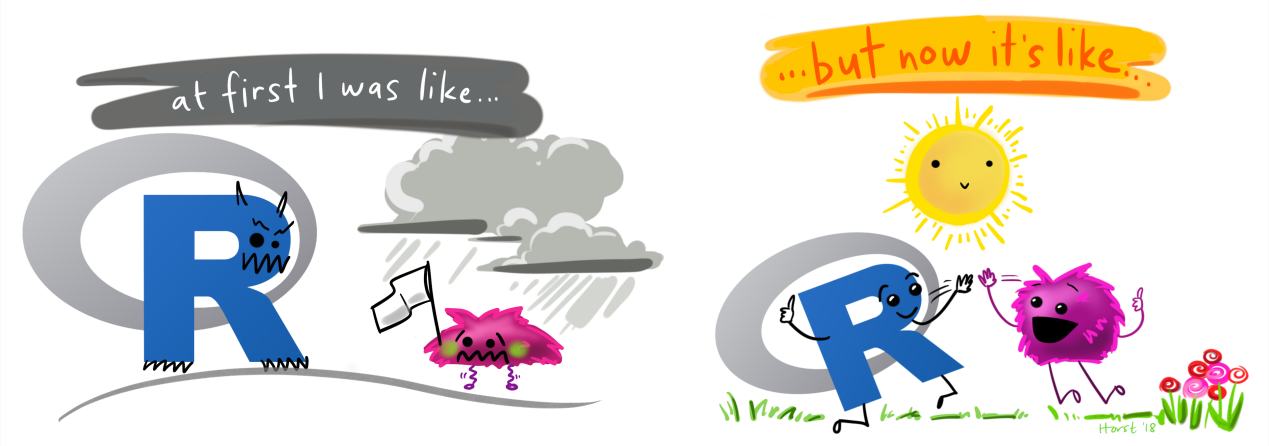
Who can write a package?

- Can you open and run R / RStudio?
- Can you install a package?
- Can you write R code?
- Can you write an R function?

Who can write a package?

- Can you open and run R / RStudio?
- Can you install a package?
- Can you write R code?
- Can you write an R function?
- Can you *learn* to write an R function?

→ **You can write a package in R**



[illustration by @allison_horst]

Before we start..

Creating packages is a process

- There's no need to do everything at once!
- Come up with milestones and focus on reaching them
- Ask yourself: what's the main purpose of your package?
- Should your package be about ...
    - Data Visualization
    - Implementation of statistical models
    - Wrapping an existing API
    - Data sharing
    - ...

Before we start..

Creating packages is a process

- There's no need to do everything at once!
- Come up with milestones and focus on reaching them
- Ask yourself: what's the main purpose of your package?
- Should your package be about ...
    - Data Visualization
    - Implementation of statistical models
    - Wrapping an existing API
    - Data sharing
    - ...

Setup

- make sure you installed the latest R and RStudio version
- install the following packages:

```
pkgs <- c("devtools", "roxygen2", "usethis", "testthat")
install.packages(pkgs)
```

Choosing a Package Name

The `available` package will help check whether your desired package name is still available:

```
library(available)
available("missingstats", browse = FALSE)
```

```
Urban Dictionary can contain potentially offensive results,
  should they be included? [Y]es / [N]o:
1: Y
── missingstats ──────────────────────────────────────────────────────────
Name valid: ✓
Available on CRAN: ✓
Available on Bioconductor: ✓
Available on GitHub:  ✓
Abbreviations: http://www.abbreviations.com/missingstats
Wikipedia: https://en.wikipedia.org/wiki/missingstats
Wiktionary: https://en.wiktionary.org/wiki/missingstats
Urban Dictionary:
  Not found.
Sentiment:???
```

Choosing a Package Name

That feeling when your chosen package name is still available ...

Our first R package!

Using `usethis` to create R packages

The purpose of `usethis` is to

> *... automate repetitive tasks that arise during project setup and development, both for R packages and non-package projects.*

→ `usethis` is *very* useful for package creation

Using `usethis` to create R packages

The purpose of `usethis` is to

> *... automate repetitive tasks that arise during project setup and development, both for R packages and non-package projects.*

→ `usethis` is *very* useful for package creation

Main functions

- two types of (main) functions within `usethis` (although there are *many* more, see the *documentation*)
  - `use_*` (for example `use_pipe` to include the pipe operator in your package)
  - `create_*` (for example `create_from_github` which creates a local Git repository from GitHub)

Creating your first R Package

The following code will create a minimal R package (in your home directory):

```
library(usethis)
create_package("~/missingstats")
```

- you only need to specify a path
  - if it exists, it is used
  - if it does not exist, it is created, provided that the parent path exists

Creating your first R Package

The following code will create a minimal R package (in your home directory):

```
library(usethis)
create_package("~/missingstats")
```

- you only need to specify a path
  - if it exists, it is used
  - if it does not exist, it is created, provided that the parent path exists
- however, it is recommended to use `create_tidy_package` instead:

```
create_tidy_package("~/missingstats")
```

- will also create a new package *but* it will also apply many great conventions that will come in handy
- note that a new R project is created and opened automatically

Creating your first R Package

This is how your working directory should look like after creating a tidy package:

```
-- R
-- tests
-- DESCRIPTION
-- NAMESPACE
-- missingstats.Rproj
-- cran-comments.md
-- LICENSE.md
-- README.md
-- README.Rmd
```

Creating your first R Package

Congratulations! You created your very first R package!

Adding functions to the package

Naming Functions

The `rOpenSci` package guide states that:

> *Functions and arguments naming should be chosen to work together to form a common, logical programming API that is easy to read, and auto-complete.*

Naming Functions

The `rOpenSci` package guide states that:

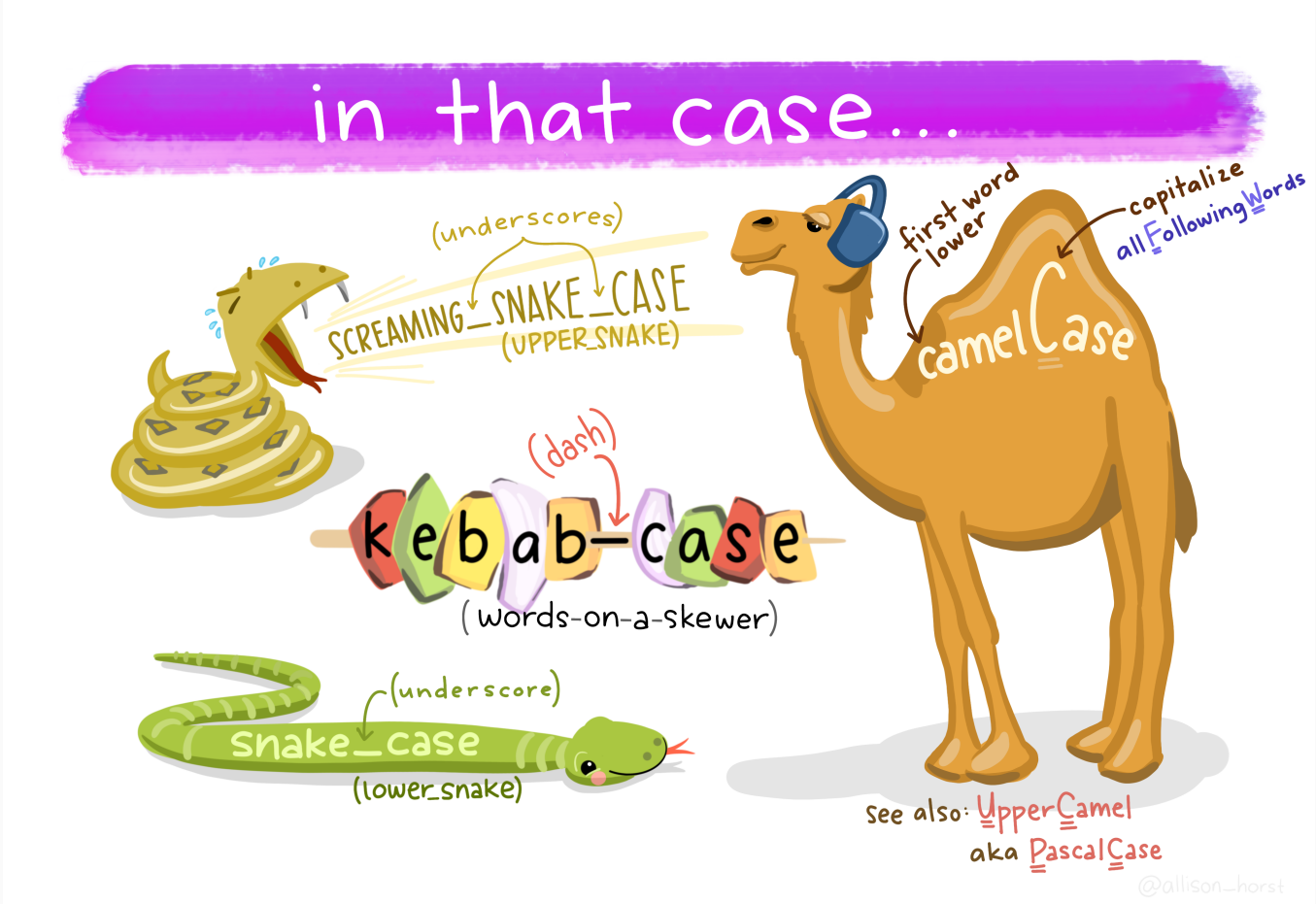
> *Functions and arguments naming should be chosen to work together to form a common, logical programming API that is easy to read, and auto-complete.*

So in the best case scenario you come up with a naming scheme in the following style:

- `object_verb`
  - helps avoid namespace conflicts with packages that may have similar verbs
  - makes code readable and easy to auto-complete
- Example: `stringr` functions all start with `str_*`

(In the examples here we won't be using this naming style just for convenience but in your personal package you should absolutely consider using it!)

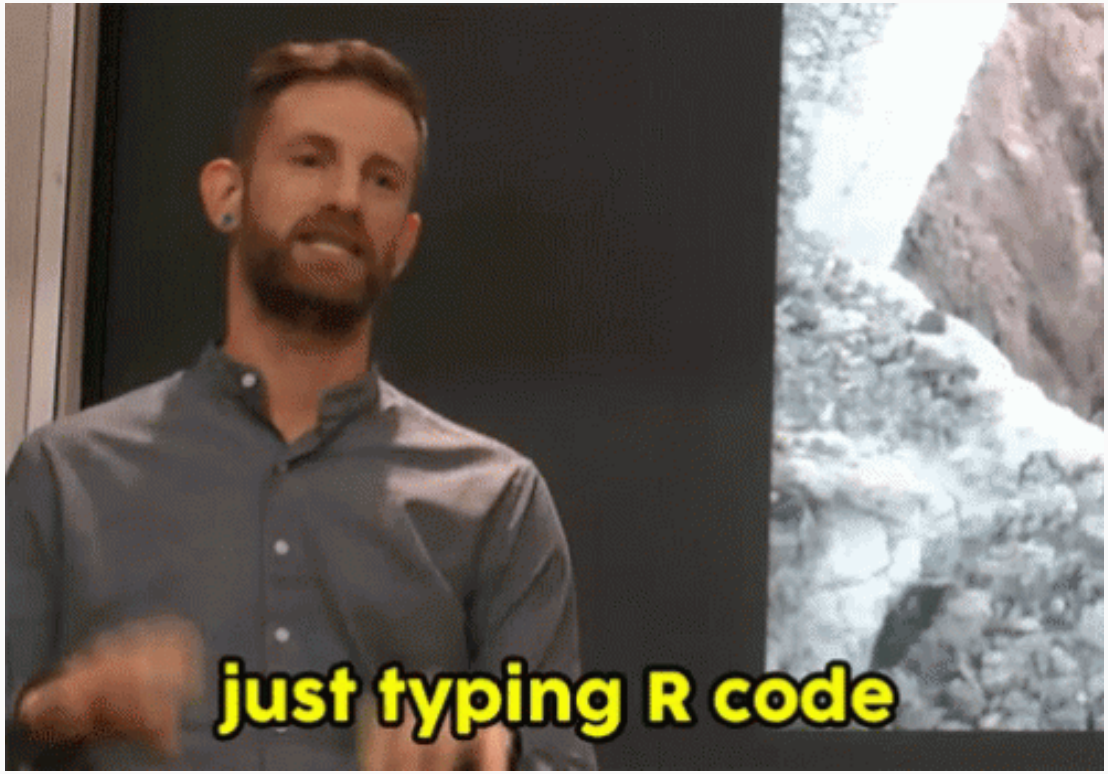
Naming Functions



[illustration by @allison_horst]

Creating your first function in your package

It’s time to type some R code!



Creating your first function in your package

- to get started we will create a new R script with the help of the `usethis` package:

```
usethis::use_r("vhb")
```

- this will create and open a `vhb.R` file in your package's R subfolder
  - you can now put all the functions of your package will in this `.R` file
  - or create several `.R` files for different (major) functions / parts of your package

Creating your first function in your package

- to get started we will create a new R script with the help of the `usethis` package:

```
usethis::use_r("vhb")
```

- this will create and open a `vhb.R` file in your package's R subfolder
  - you can now put all the functions of your package will in this `.R` file
  - or create several `.R` files for different (major) functions / parts of your package

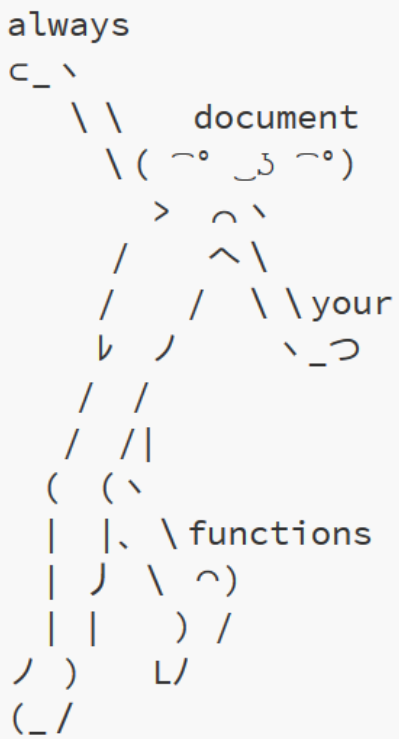
Let's create an easy function that calculates the standard error of the mean: $\sqrt{\frac{\mathrm{Var}}{n}}$

```
se <- function(x) {
  sqrt(var(x)/length(x))
}

se(c(0,6,9))
## [1] 2.645751
```

Documentation

How can we communicate what functions do?



Documentation

Documentation is absolutely central!

- basically the guidance for your users so they know how to use the functions within your package

As Hadley Wickham puts it:

> *Documentation is one of the most important aspects of good code. Without it, users won't know how to use your package, and are unlikely to do so.*

Documentation

Documentation is absolutely central!

- basically the guidance for your users so they know how to use the functions within your package

As Hadley Wickham puts it:

> *Documentation is one of the most important aspects of good code. Without it, users won't know how to use your package, and are unlikely to do so.*

Package documentation in R

- built-in functionality within R to document packages: `.Rd` files (stored in the `man` subdirectory of your package)
  - a syntax that is similar to latex
- package `roxygen2` conveniently creates a lot of the necessary files with its own more intuitive syntax style

Documentation with roxygen2

This is how the documentation for our `se` function might look like using roxygen2:

```
#' Estimate the standard error of the mean.
#'
#' @param x A vector of numbers.
#' @return The standard error of of the mean regarding \code{x}.
#' @export
#' @examples
#' se(1:5)
#' se(c(0,6,9))
se <- function(x) {
  sqrt(var(x)/length(x))
}
```

That is, to document a function, just put the roxygen documentation directly above your function!

Documentation with `roxygen2`

se {missingstats} R Documentation

Estimate the standard error of the mean.

Description

Estimate the standard error of the mean.

Usage

```
se(x)
```

Arguments

`x`	A vector of numbers.

Value

The standard error of of the mean regarding `x`.

Examples

```
se(1:5)
se(c(0,6,9))
```

Documentation with `roxygen2`

Converting the `roxygen` description to `.Rd` files

- to appear in the help file, the `roxygen` code has to be converted to a `.Rd` file so it can appear in the help file
- with the help of the `devtools` package this step is *really* easy to do
  - read more about object documentation in the *object documentation* chapter from Hadley's *R packages* book

```
# convert roxygen description to .Rd
devtools::document()
```

```
Updating missingstats documentation
Writing NAMESPACE
Loading missingstats
Writing NAMESPACE
Writing vhb.Rd
```

→ typing `?se` into R it should render the development description from the previous slide

Automatic Tests (Unit Testing)

Unit testing

Why unit testing

- testing is important to make sure your package is functioning as intended
  - as you keep developing and adding to your package code, keeping track of what might go wrong will become more and more complex
- unit tests help with identifying issues within your code (to pinpoint what went wrong and where)

Unit testing

Why unit testing

- testing is important to make sure your package is functioning as intended
    - as you keep developing and adding to your package code, keeping track of what might go wrong will become more and more complex
- unit tests help with identifying issues within your code (to pinpoint what went wrong and where)

Logic of unit testing

- write up example code with your package functions and define specific outputs that you expect
    1. If they are met, great, the test is passed!
    2. If not, the test fails and you will know that something is wrong with your code and what failed.

Unit testing with `testthat`

How to set up unit tests

- package `testthat` to set up unit tests for your package
- simply run `use_testthat` once (from the `usethis` package)

```
usethis::use_testthat()
```

→ creates a folder ‘tests‘ and a subfolder ‘testthat‘ (where your tests will live)

(because we created our package with `create_tidy_package` in the beginning, `testthat` is automatically set up and we don’t need this step)

Unit testing with `testthat`

Setting up individual tests

```
# package `usethis` again helpful
usethis::use_test("vhb")
```

This will create the following file:

```
tests/testthat/test-vhb.R
```

With this content:

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

Unit testing of your own functions

You ready?

Unit testing of your own functions

Define your own unit tests

- define positive (expected) outcomes

```
library(testthat)
test_that("standard error works as intended", {
  result <- se(c(2,6))
  expect_equal(result, 2)
})
```

→ the result should be *silent* (that is, we pass the test and we get no error)

Unit testing of your own functions

- also define negative outcomes (where your function should fail)
- be precise in what you expect! (i.e., do not just write “this should not work” but “this should throw an error”)

Unit testing of your own functions

- also define negative outcomes (where your function should fail)
- be precise in what you expect! (i.e., do not just write “this should not work” but “this should throw an error”)

```
# be precise: we expect an error
test_that("supplying string should throw error", {
  expect_error(
    se(c("one", 2, 3))
    )
})
## Error: Test failed: 'supplying string should throw error'
## * `se(c("one", 2, 3))` did not throw an error.
```

→ the test fails (because we only get a warning)

Unit testing of your own functions

- also define negative outcomes (where your function should fail)
- be precise in what you expect! (i.e., do not just write “this should not work” but “this should throw an error”)

```
# be precise: we expect an error
test_that("supplying string should throw error", {
  expect_error(
    se(c("one", 2, 3))
    )
})
## Error: Test failed: 'supplying string should throw error'
## * `se(c("one", 2, 3))` did not throw an error.
```

→ the test fails (because we only get a warning)

```
test_that("supplying string should throw warning", {
  expect_warning(
    se(c("one", 2, 3))
    )
})
```

Unit testing of your own functions

Testing all tests at once

- we have successfully written up three tests for our `se` function
- to test all unit tests at once, run the `test()` command from `devtools` package

```
devtools::test()
```

```
Loading missingstats
Testing missingstats
v |  OK F W S | Context
x |   2 1 1   | vhb
──────────────────────────────────────────────────────────────────────────────
test-vhb.R:7: warning: supplying string should throw error
NAs introduced by coercion

test-vhb.R:7: failure: supplying string should throw error
`se(c("one", 2, 3))` did not throw an error.
──────────────────────────────────────────────────────────────────────────────

══ Results ═════════════════════════════════════════════════════════════════
Duration: 0.1 s

OK:       2
Failed:   1
Warnings: 1
Skipped:  0
```

Including Data in R Packages

Some R packages only include data

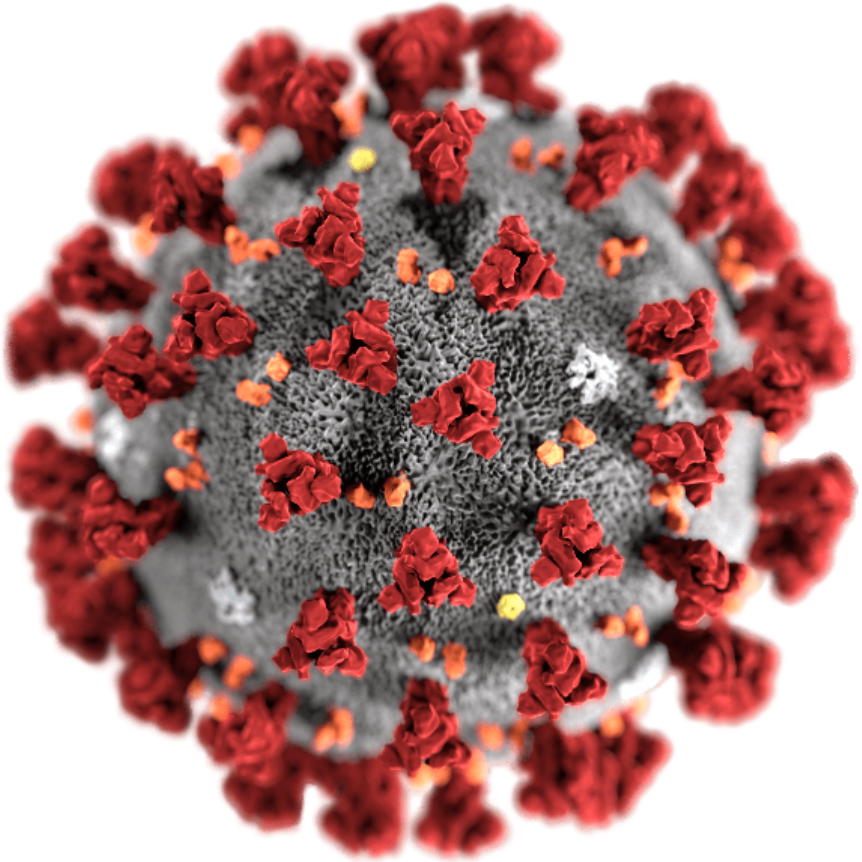
coronavirus

build failing | CRAN 0.1.0 | lifecycle experimental | License MIT

The coronavirus package provides a tidy format dataset of the 2019 Novel Coronavirus COVID-19 (2019-nCoV) epidemic. The raw data pulled from the Johns Hopkins University Center for Systems Science and Engineering (JHU CCSE) Coronavirus repository.

More details available here, and a `csv` format of the package dataset available here

A summary dashboard is available here



How to include (external) data in R packages

Raw data / Data creation

- if you are going to include data in your package, you might want to include a `data-raw` folder where the data is created/formatted
  - e.g., the code that was used to scrape web data
  - e.g., the code that was used to generate higher-level (aggregated) data

How to include (external) data in R packages

Raw data / Data creation

- if you are going to include data in your package, you might want to include a `data-raw` folder where the data is created/formatted
  - e.g., the code that was used to scrape web data
  - e.g., the code that was used to generate higher-level (aggregated) data
- just use `use_data_raw()` to initialize everything in the right place
  - creates the `data-raw` folder (and an exemplary `DATASET.R` script in `data-raw`)

```
usethis::use_data_raw()
```

```
✓ Creating 'data-raw/'
✓ Adding '^data-raw$' to '.Rbuildignore'
✓ Writing 'data-raw/DATASET.R'
● Modify 'data-raw/DATASET.R'
● Finish the data preparation script in 'data-raw/DATASET.R'
● Use `usethis::use_data()` to add prepared data to package
```

How to include (external) data in R packages

Adding prepared data

- use `usethis::use_data()` to add prepared data to your package (they go into the `data` folder)
  - creates `.rda` files in `data/`
  - use option `compress = "xz"` for large datasets

How to include (external) data in R packages

Adding prepared data

- use `usethis::use_data()` to add prepared data to your package (they go into the `data` folder)
  - creates `.rda` files in `data/`
  - use option `compress = "xz"` for large datasets

```
babynames <- c("Emma", "Liam", "Alex")
cities <- c("Frankfurt", "Cologne")
```

```
usethis::use_data(babynames, cities)
```

```
✓ Creating 'data/'
✓ Saving 'babynames', 'cities' to 'data/babynames.rda', 'data/cities.rda'
```

Documenting data

How can you document your data?

- by convention, create a single `data.R` file (using `use_r("data")`) that contains the information for all datasets of your package

Documenting data

How can you document your data?

- by convention, create a single `data.R` file (using `use_r("data")`) that contains the information for all datasets of your package
- use `roxygen` to document your data
  - as an example, have a look at the data documentation in the `babynames` package

```
#' Baby names.
#'
#' Full baby name data provided by the SSA. This includes all names with at
#' least 5 uses.
#'
#' @format A data frame with five variables: \code{year}, \code{sex},
#'   \code{name}, \code{n} and \code{prop} (\code{n} divided by total number
#'   of applicants in that year, which means proportions are of people of
#'   that sex with that name born in that year).
"babynames"
```

The final check

Checking the package

Bringing it all together

- run `check()` from `devtools` to test pretty much everything about our package

```
devtools::check()
```

```
── Checking ──────────────────────────────────────────────────── missingstats ──
Setting env vars:
● _R_CHECK_CRAN_INCOMING_REMOTE_: FALSE
● _R_CHECK_CRAN_INCOMING_       : FALSE
● _R_CHECK_FORCE_SUGGESTS_      : FALSE
── R CMD check ─────────────────────────────────────────────────────────────────
─  using log directory 'C:/Users/stefan/AppData/Local/Temp/RtmpWS1O3S/missingstats.Rcheck' (393ms)
─  using R version 3.6.1 (2019-07-05)
─  using platform: x86_64-w64-mingw32 (64-bit)
─  using session charset: ISO8859-1
─  using options '--no-manual --as-cran'
√  checking for file 'missingstats/DESCRIPTION' ...
─  this is package 'missingstats' version '0.0.0.9000'
─  package encoding: UTF-8
√  checking package namespace information ...
√  checking package dependencies (1.3s)
√  checking if this is a source package ...
√  checking if there is a namespace
√  checking for .dll and .exe files
√  checking for hidden files and directories ...
√  checking for portable file names
√  checking serialization versions ...
√  checking whether package 'missingstats' can be installed (2.6s)
√  checking package directory
√  checking for future file timestamps (475ms)
√  checking DESCRIPTION meta-information (631ms)
√  checking top-level files
√  checking for left-over files ...
√  checking index information
```

Checking the package

And when your package passes all checks you get the sweet feeling of getting the following feedback ...

Further resources

More useful concepts for R package development

Importing / Relying on other packages

- add Package imports to `DESCRIPTION` with `usethis::use_package()`

```
# if you rely / use dplyr
usethis::use_package("dplyr")
```

More useful concepts for R package development

Importing / Relying on other packages

- add Package imports to `DESCRIPTION` with `usethis::use_package()`

```
# if you rely / use dplyr
usethis::use_package("dplyr")
```

Continuous integration (CI)

- set up continuous integration (in this case Travis CI) with `usethis::use_travis()`
  - allows to run checks on your package code with various R versions
  - ensures that your package can run on different systems like Linux, Mac OSX & Windows
- read more about Travis CI in *Julia Silge's Blog*

```
usethis::use_travis()
```

More useful concepts for R package development

Package websites

- set up a website for your package (documentation) with the help of **pkgdown** (see its own *documentation*)

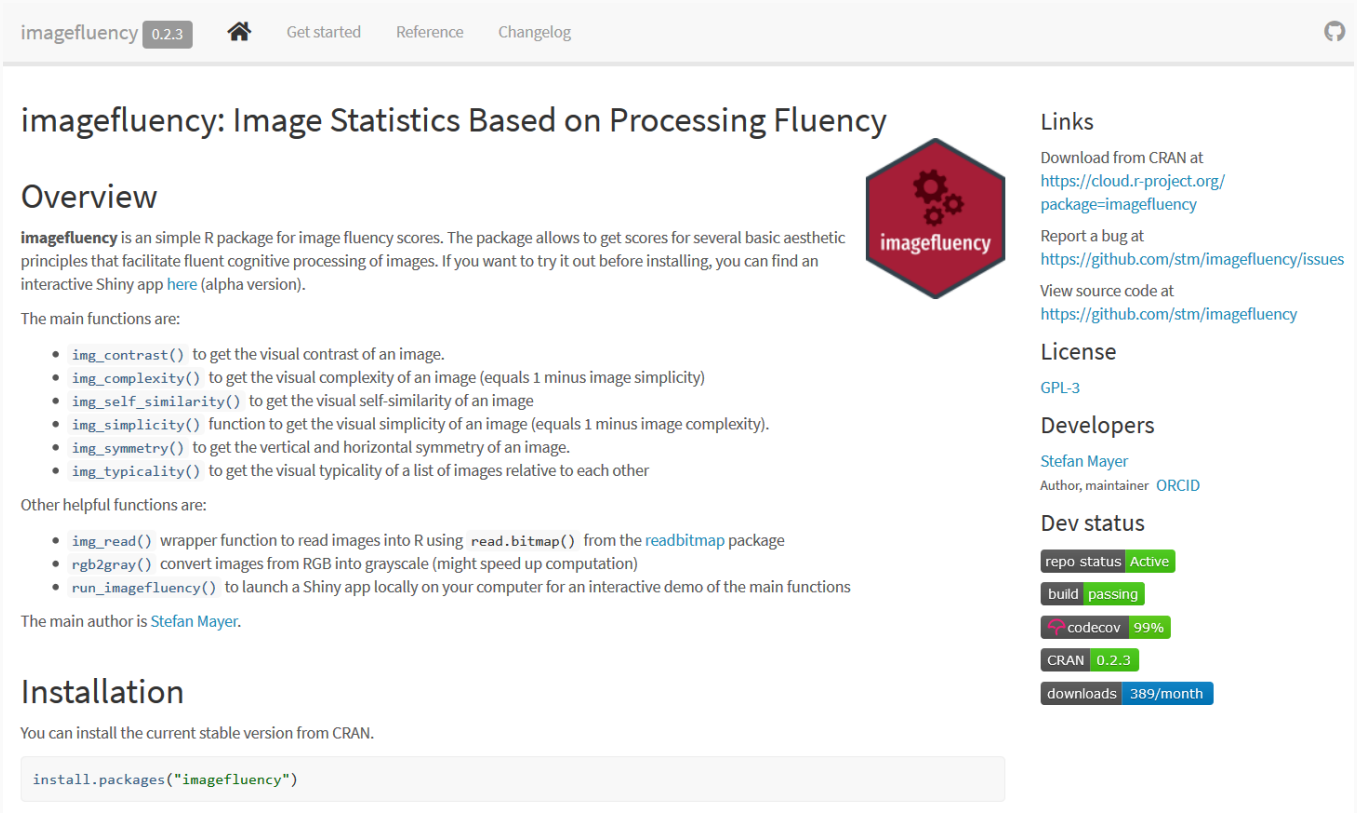
```
usethis::use_pkgdown()
```

More useful concepts for R package development

Package websites

- set up a website for your package (documentation) with the help of **pkgdown** (see its own *documentation*)

```
usethis::use_pkgdown()
```



Further resources

R package development

- Hadley Wickham's *R Packages Book*
- R-Ladies Baltimore's *Blog Post*
- Emil Hvitfeldt's *Blog post about usethis for package development*
- RStudio's *Support Website*
- rOpenSci's *Package Guide*

Data sharing with R

- Hadley's *Book Chapter about Data*
- Bruno Rodrigues' *NetHack Tutorial*
- Dan Quintana's *Primer* about the `synthpop` *package*

Note: This presentation was heavily inspired by Fabio Votta's *Package Development in R* slides.

Thank you for your attention.

Stefan Mayer
School of Business and Economics
University of Tübingen
stefan.mayer@uni-tuebingen.de