



Workshop

Building R Packages



Agenda

-  Introductions
-  Package setup
-  Development cycle
-  Testing
-  Good practice





Mango Solutions

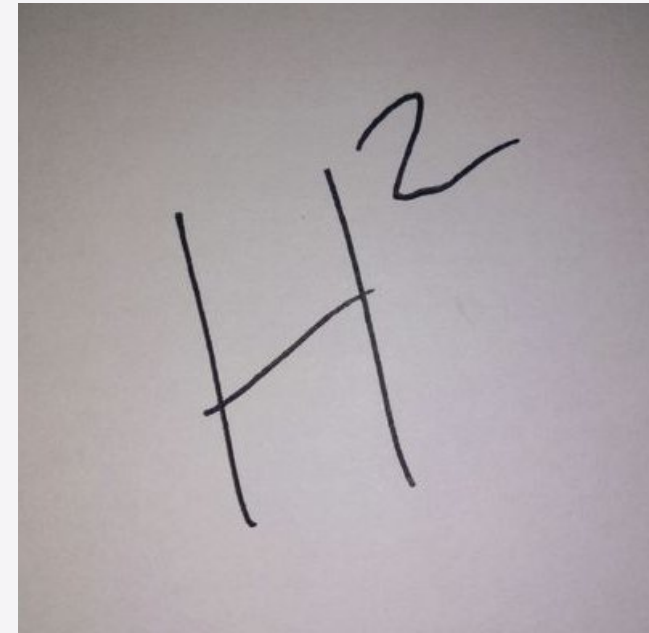
- Data science services
- Cross sector
- ~ 70 people
- London and Chippenham
- We ❤️ R, Python and Spark





Hello, it's me!

-  Data Scientist @ Mango
-  PhD in statistics
-  R-Ladies
-  R packages: psychomix, trackeR, goodpractice
-  Twitter: @hfcfrick





Why make an R package?

One place for your code, documentation and tests means you can

- Simplify loading of code and packages
- Maintain a single version and are able to identify which version is being used
- Implement a testing framework to more confidently make changes to the code
- Provide documentation and usage examples of code easily

Who to make an R package for

- Yourself only
- Your team / internal use
- The wider public



Resources

Online

- Book: R Packages.
 - 1st edition by Hadley Wickham at <http://r-pkgs.had.co.nz/>,
 - 2nd edition with Jenny Bryan is work-in-progress at <https://r-pkgs.org/>
- Cheatsheet for devtools at <https://www.rstudio.com/resources/cheatsheets/>
- Writing R Extensions by R Core at <https://cran.r-project.org/doc/manuals/r-release/R-exts.html>

In person

Mango offers 1-day workshops on package building and plenty of other topics such as text mining and web scraping, various levels of shiny, functional programming,



devtools and friends

The devtools package is turning into a meta package and includes, e.g.,

- `usethis`: Automating package setup
- `pkgload`: Simulate package loading
- `pkgbuild`: Building binary packages
- `roxygen2`: Function and package documentation
- `rcmdcheck`: Running R CMD check
- `testthat`: Writing and running tests





Components of an R package

Essential

- DESCRIPTION
- NAMESPACE
- R/
- man/

Optional

- tests/
- vignette/
- inst/
- data/
- src/
- ...



Getting started

Via Code

```
create_package("path/to/yourpackage")
```

Via RStudio

New Project > New Project > R Package



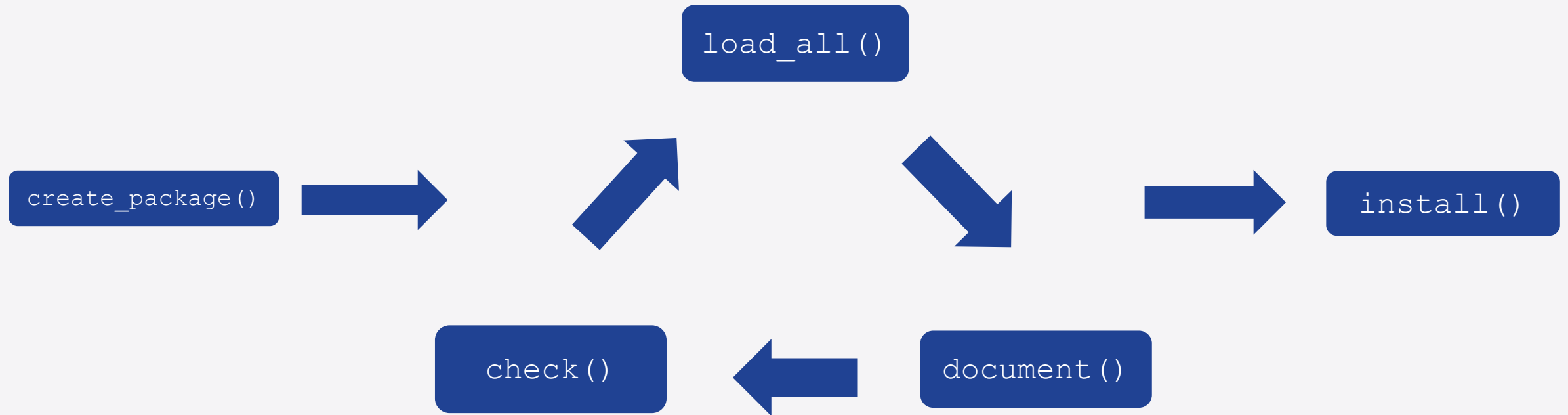
Exercise

Adapted from Chapter 2 *The Whole Game* of R Packages [<https://r-pkgs.org/whole-game.html>]

- Create a new package, call it *foofactors* or anything you like



Typical workflow





Code development

`load_all()`

- `load_all()` simulates building, installing and attaching your package. Use it to interactively develop your code
- Keep your code simple!
- Make your code more readable for yourself and others (or just your future you) by using a consistent style
- Popular style guide: <https://style.tidyverse.org/>
- Packages supporting this style guide:
 - `styler` to restyle code interactively
 - `lintr` to automatically check



Exercise

Adapted from Chapter 2 *The Whole Game* of R Packages [<https://r-pkgs.org/whole-game.html>]

- Create a new package, call it *foofactors* or anything you like
- Add a .R file with a short function, e.g.,

```
fbind <- function(a, b) { factor(c(as.character(a), as.character(b))) }
```



Check your package

check ()

Minimal requirements for distribution via CRAN are assessed via command line tool R CMD check. This checks, e.g.,

- Documentation for all functions available to the user
- Dependencies on other R packages
- Examples

Run all these checks from within your R session with `check ()`.



Exercise

Adapted from Chapter 2 *The Whole Game* of R Packages [<https://r-pkgs.org/whole-game.html>]

- Create a new package, call it *foofactors* or anything you like
- Add a .R file with a short function, e.g.,

```
fbind <- function(a, b) { factor(c(as.character(a), as.character(b))) }
```

- Check your package



DESCRIPTION

Edit the template

- Title and description
- Author(s)
- Licence, e.g., via `use_*_license()` such as `use_mit_license()`
- Dependencies, e.g., via `use_package()`



Exercise

Adapted from Chapter 2 *The Whole Game* of R Packages [<https://r-pkgs.org/whole-game.html>]

- Create a new package, call it *foofactors* or anything you like
- Add a .R file with a short function, e.g.,

```
fbind <- function(a, b) { factor(c(as.character(a), as.character(b))) }
```

- Check your package
- Fix your DESCRIPTION



Documentation

`document ()`

- Keep documentation in the same place, easier to spot if you changed the name or order of your arguments, etc
- Insert roxygen skeleton via *Code > Insert roxygen skeleton* or shortcut *Shift+Alt+Cmd/Ctrl+ R*
- `document ()`
 - Creates man/ folder and .Rd files
 - Updates NAMESPACE



Documentation

Roxygen2 syntax

- `@param`
- `@return`
- `@examples`
- `@export`
- `@import` / `@importFrom`



Exercise

Adapted from Chapter 2 *The Whole Game* of R Packages [<https://r-pkgs.org/whole-game.html>]

- Create a new package, call it *foofactors* or anything you like
- Add a .R file with a short function, e.g.,

```
fbind <- function(a, b) { factor(c(as.character(a), as.character(b))) }
```

- Check your package
- Fix your DESCRIPTION
- Add documentation for your function and check your package again



Build and install your package

build()

- Builds tarball by default
- Can also build binary package with `binary = TRUE`

install()

- Does what it says on the tin



Testing

Motivation

- Prevent bugs
- Formalise ad-hoc testing
- Refactor with confidence
- Allows you to notice and locate bugs faster

Workflow

- Each bug should get its own test
- Each repeated interactive testing on the console should get its own test



Testing with testthat

Infrastructure

- Setup via `use_testthat()`
- Create test via `use_test()`: test files live in `tests/testthat` and their name starts with `test_`
- Run tests via `test()`
- Structure of a test file: context > tests > expectations



Testing with testthat

Example

```
context("Binding factors")

test_that("fbind() binds factor (or character)", {
  x <- c("a", "b")
  x_fact <- factor(x)
  y <- c("c", "d")
  z <- factor(c("a", "b", "c", "d"))
  expect_identical(fbind(x, y), z)
  expect_identical(fbind(x_fact, y), z)
})
```





Exercise

Adapted from Chapter 2 *The Whole Game* of R Packages [<https://r-pkgs.org/whole-game.html>]

- Create a new package, call it *foofactors* or anything you like
- Add a .R file with a short function, e.g.,

```
fbind <- function(a, b) { factor(c(as.character(a), as.character(b))) }
```

- Check your package
- Fix your DESCRIPTION
- Add documentation for your function and check your package again
- Set up testing infrastructure and add a test for your function, e.g., the code from the example



Test coverage with covr

Test coverage: Share of your code covered with tests

- Check coverage via `package_coverage()`
- Inspect coverage via `report()`





Writing user guides (vignettes)

- Setup via `use_vignette("my-vignette")`. This does the following:
 - Create a vignettes/ folder
 - Add the necessary dependencies to the DESCRIPTION
 - Draft a vignette, vignettes/my-vignette.Rmd
- Long-form guide to your package: Explain the purpose of your package and how to use it!
- Markdown syntax



Good practice for package development

Essential

- Make sure the CRAN checks pass
- Use tests – test (most of) your code
- Use a coding standard for readability
- Keep your functions simple



Good practice for package development

Bonus points

- Make it user-friendly, e.g., add a vignette
- Keep track of changes via version control
- Use automation
- Make your code public



The goodpractice package

- Run via `goodpractice("path/to/pkg")` or its alias `gp()`
- Performs static code analysis covering the essential bases
 - CRAN checks
 - Test coverage
 - Linting
 - Cyclomatic complexity
- Use it for advice or as a reminder (to yourself or others)
- Configurable and extensible: make it yours!





Exercise

Adapted from Chapter 2 *The Whole Game* of R Packages [<https://r-pkgs.org/whole-game.html>]

- Create a new package, call it *foofactors* or anything you like
- Add a .R file with a short function, e.g.,

```
fbind <- function(a, b) { factor(c(as.character(a), as.character(b))) }
```

- Check your package
- Fix your DESCRIPTION
- Add documentation for your function and check your package again
- Set up testing infrastructure and add a test for your function, e.g., the code from the example
- Check your package with `goodpractice::gp()`



Tips and tricks

- *Use version control*
- Create a website for your package with the **pkgdown** package
- Check available package names with the **available** package
- Check your package on different platforms on r-hub with the **rhub** package
- Dig through the functionality of the **usethis** package to discover, e.g.,
 - `use_pipe()` for using the pipe operator `%>%` in your package
 - `use_readme_rmd()` and `use_readme_md()` for readmes, e.g., on GitHub