How to create your own R package

Statistics, Visualization and More Using "R"

Marlene Brunner & Melanie Löckinger

Paris Lodron Universität Salzburg

13. Juni 2022

Agenda

Motivation - Why to Create a Package?

Functions are a way to don't repeat yourself and be more efficient.
You can share workflows and empower yourself and your team.
You can test your code and "trust your work".
Ultimately, functions make your work much easier, faster, and more reproducible
... and R packages let you share these functions and be lazier, in a good way!

Motivation - Why to Create a Package?

Functions are a way to don't repeat yourself and be more efficient.
You can share workflows and empower yourself and your team.
You can test your code and "trust your work".
Ultimately, functions make your work much easier, faster, and more reproducible
... and R packages let you share these functions and be lazier, in a good way!

- copy & paste
- recode same code when reusing
- standardize your code
- consistent

IT IS MUCH EASIER THANK YOU THINK

What Is a Package?

A package is "a home for functions."
Functions are a home for source code.
Packages are a way of **describing** and **distributing** these functions, in a **structured** and **consistent** way.

What Is a Package?

A package is "a home for functions."
Functions are a home for source code.
Packages are a way of **describing** and **distributing** these functions, in a **structured** and **consistent** way.
There are five different states of a package:

- installed
- in-memory
- binary
- bundled
- source

Let's Get Started - A Cookbook

1. start R Studio
   no file, just the console
2. load the packages

```
# install.packages(devtools)
# install.packages(usethis)
library(devtools)
library(usethis)
```

   devtools: makes package development easier by providing R functions that simplify and expedite common tasks.
   usethis: workflow package that automates repetitive tasks that arise during project setup and development, both for R packages and non-package projects.
3. 'new project' → 'new directory' → 'R package'

Anatomy of a New Package

- metadata via the **DESCRIPTION**, including the name of the package, description of the package, the version of the package, and any package dependencies.
- source code via **.R** files, that live in the **R** directory.
- special roxygen comments inside the .R files that describe how the function operates and its arguments, dependencies, and other metadata
- the **NAMESPACE** for the exported functions you have written, and the imported functions you bring in
- **tests** that confirm your function "works as intended"
- **R.buildignore**: specify files that you need during package development, but not part of final package
- **man**: 'manual' – holds documentation
- **NAME.Rproj**: we don't "need" it; that's an Rstudio file for opening your project
- other things (installed files, compiled code, data, tutorials, vignettes)

Again - But Without Predefined Functions

'new project' $\rightarrow$ 'new directory' $\rightarrow$ 'R package' (with devtools)

we can now fill the empty box with functions

Step by Step

- Step 1: create the function

```
usethis::use_r('NewFunctionName.R')
```

this opens an R file

- Step 2: write the function

```
square_value <- function(x){
x^2
}
```

- Step 3: load the function(s)

```
devtools::load_all()
```

- Step 4: check the function

```
devtools::check
```

License Warning

providing information about the license of your package is part of the CRAN policy

```
usethis::use_gpl_license(version = 3, include_future = TRUE)
```

adds an open source license to your description file - there are many others

Documentation With roxygen2

We will describe our functions in comments next to their definitions. roxygen2 will process our source code and comments and automatically generate .Rd files in

- man/
- NAMESPACE
- and if needed, the Collate field in DESCRIPTION

roxygen items are indicated with special comments (#'), e.g.:

```
#' @param argument A numeric input, that will be squared
```

click into function → Code → Insert Roxygen Skeleton

Documentation

- Title : title of the function
- description of the function purpose
- @param: documenting the function arguments/parameters
- @return: What does the function return?
- @export:
- @examples: add an example (might be helpful)
  important: example has to work

```
devtools::document()
```

Installation

```
devtools::install()
```

As soon as the installation is done, you can call your function in any environment / package:

```
library(PackageName)
PackageName::FunctionName()
```

and examine the manual for your package:

```
help(package=PackageName)
```

→ adapt your DESCRIPTION

Additional Comments

- never use library() or require() inside R code

```
use_package(package, type = "Imports", min_version = NULL)
```

  will add your dependencies to the description file
- return() not strictly needed
- constant health checks!

Do Your Own Functions

1. `usethis::use_r(‘functionName.R‘)`
2. write your function
3. `devtools::load_all()`
4. `devtools::check()`
5. Documentation with ‘Insert Roxygen Skeleton‘

   `devtools::document()`
6. `devtools::install()`

Who wants to show something?

Example

```r
#' Bootstrap Confidence Interval for Difference in Sample Means
#'
#' Takes two samples, the number of bootstrap samples to generate and a significance level.
#' The difference of the bootstrap sample means will be computed for each bootstrap sample.
#' The resulting bootstrap distribution provides the percentiles for the confidence interval.
#'
#' @param x sample 1
#' @param y sample 2
#' @param R number of runs
#' @param alpha significance level of the confidence interval
#'
#' @return the confidence interval based on the percentiles of the bootstrap
#' distribution for the difference in sample means
#' @export
#'
#' @examples ci(rnorm(100), rnorm(200), 1000, 0.05)
ci <- function(x, y, R, alpha){
    boot.diff <- rep(0, R)
    for (i in 1:R) {
        x.boot <- sample(x, size=length(x), replace=TRUE)
        y.boot <- sample(y, size=length(y), replace=TRUE)
        boot.diff[i] <- mean(x.boot) - mean(y.boot)
    }
    ci.boot <- as.numeric(stats::quantile(boot.diff,probs = c(alpha/2,1-alpha/2)))
    return(ci.boot)
}
```

Testing I

You can always try your functions in the console.

Up until now, we've only tested our function interactively and checked for package errors via check().
We can formalize and expand this with some unit tests via testthat.

Testing I

You can always try your functions in the console.

Up until now, we've only tested our function interactively and checked for package errors via check().
We can formalize and expand this with some unit tests via testthat.
Why?

- Fewer bugs
- Better code structure
- Easier restarts
- Robust code

Testing II

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

- create a tests/testthat directory.
- add testthat to the Suggests field in the DESCRIPTION.
- create a file tests/testthat.R that runs all your tests

Test your package with

```
usethis::use_test()
```

Testing III (Expectations)

```
expect_equal(
  current,
  target,
  tolerance = sqrt(.Machine$double.eps),
  info = NA_character_,
  ...
),
expect_identical(), expect_match()
expect_output()
expect_message()
expect_warning(), expect_error()
expect_is()
expect_true(), expect_false()
```

Testing IV (Expectations)

```
library(testthat)

test_that("computation works", {
    expect_equal(square_value(2), 2^2)
    expect_equal(square_value(3), 3^2)
    })

test_that("Non-numeric or missing inputs should error", {
    expect_error(square_value("a"), "non-numeric argument to
    binary operator")
    })
```

Do Your Own Test

Write your own test for your individual function. Don't forget to actually try your test.

```
library(testthat)

...
```

https://r-pkgs.org/tests.html#expectations

Vignette

- long-form guide to your package
- divides functions into useful categories
- demonstrate how to coordinate multiple functions to solve problems
- provides three things:
  - the original source file
  - a readable HTML page or PDF
  - a file of R code

```
browseVignettes()
browseVignettes("packagename")
```

https://cran.r-project.org/web/packages/dplyr

Some More Code I

```
usethis::use_version()
```

to increment your package version (changes DESCRIPTION), note that this commits to git as well!

```
usethis::use_data()
```

to add data; e.g.

```
usethis::use_data(iris)
```

Some More Code II

```
usethis::use_rcpp()
```

to use C or C++ code

```
formatR::tidy_file(file, ...)
```

to tidy any .R files

```
covr::package_coverage(type = c("tests", "vignettes", "examples",
                                "all", "none"), ...)
```

to calculate the coverage of you package, e.g. how much of your code is tested?

Sharing R Package I

You can share your package as a tar.gz "source tarball":

- platform agnostic
- can be sent around

```
devtools::build(
    pkg = ".",
    path = NULL,
    binary = FALSE,
    vignettes = TRUE,
    manual = FALSE,
    ...
)

install.packages(path_to_tar.gz, repos = NULL, type="source")
```

Sharing R Package II

You can share your package easily via Github:

- use Git version control
- RStudio project options - Version Control - local Git repository
- Git Pane
- create GitHub repository - submit

```
devtools::install_github("hadley/dplyr@v1.0.7")
```

Sharing R Package III

You can release your package via CRAN:

```
devtools::build()
devtools::release()
```

uploads package bundle to CRAN submission form → review → approved / rejected

You need ...

- version number
- README.md and NEWS.md
- checks and tests
- CRAN policies - in more detail:
  `https://github.com/eddelbuettel/crp/blob/master/txt/policies.r5236.txt`

Real Life Package

Most packages have source code on GitHub.

Activity: Look for the source code of a function you often use. Look for the manual as well.

Real Life Package

Most packages have source code on GitHub.

Activity: Look for the source code of a function you often use. Look for the manual as well.

Best Practice
`https://github.com/tidyverse/dplyr`

References

CRAN Repository Maintainers. (2022). CRAN Repository Policy. https://github.com/eddelbuettel/crp/blob/master/txt/policies.r5236.txt

R Core Team (2022). Writing R Extensions. https://cran.r-project.org/doc/manuals/r-release/R-exts.html

Vignette Example: https://cran.r-project.org/web/packages/dplyr

Wickham, H. (2015). R packages. https://r-pkgs.org/index.html

(2015). Package Development:: Cheat Sheet. https://rklopotek.blog.uksw.edu.pl/files/2017/09/package-development.pdf

Thank you for your attention!