Let's build an R-package together!

Daniel Heimgartner

May 14, 2025

1 WHY?

What is an R-package?

- You are free to organize your code however you want!
- But: There are good and bad practices (e.g., stuff like `stwd("/foo/bar")`, `install.packages("foo")`, ...)
- A package is nothing else then a standardized way to organize *information* (not necessarily code!)
- It is nothing to be afraid of!

What are the benefits?

- Code can be easily shared and is installable (and therefore loadable via `library(Rpackage)`)
- Code runs everywhere
- Code, data, documentation, paper! is organized in a comprehensible way (i.e., in a way that humans and computers understand and *expect*)
- If you conform to this structure you get a lot of additional benefits/tools for free (e.g., nicely formatted doc/man pages, easily generate a project/package/date webpage, etc.)

What will I learn?

- How to scaffold (generate) the package structure
- Where and how to put your analysis scripts
- How to write useful helper functions in separate source files (which can easily be used in other scripts – in and outside your R-package)
- How to attach data to the package
- How to document objects (package, data, functions)
- How to write your paper in Sweave, using the vignette – allowing you to intermingle R code and LaTeX
- How to share the code with your colleagues
- Where to learn more

2 LET'S GET STARTED

`usethis::create_package()`

This will scaffold the R-package. Let's run

```
usethis::create_package("Rpackage", rstudio = TRUE)
```

and navigate into the `Rpackage` folder. What did we get?

If you know git, run `usethis::use_git()`

```
.
├── DESCRIPTION
├── NAMESPACE
├── R
└── Rpackage.Rproj
```

The `.Rproj` *file is just RStudio's "project file", storing some settings regarding your project organization. In particular it opens a new RStudio session in that current folder...*

We get the minimal file and folder structure of an R-package (`Rpackage.Rproj` is optional if you are using RStudio). Congratulations – you just wrote your first package!

`usethis::use_data_raw()`

The `data-raw` folder is where the untouched raw data goes. But also all the scripts that manipulate this raw data and generate some other output (e.g., data for the analysis). I usually put all my scripts there since it allows me to easily store intermediate objects (e.g., a model fit as returned by `fit()`, see below), attach them to the package and use them at a later stage. Let's get concrete

You can also put raw data, i.e., a `.csv` *or some other file in this folder (again, it's just a folder). It is convention to name the raw data the same as the script which manipulates it, i.e.,* `useful_data.csv` *in this case...*

```
usethis::use_data_raw("useful_data")
```

This will create the `data-raw` folder and initialize the `useful_data.R` script. An example what to put there

```
#= data-raw/useful_data.R
## code to prepare `useful_data` dataset goes here
## e.g., fetch data from the qualtrics server
library(dplyr)
useful_data <- iris
## clean it
useful_data <- useful_data %>%
    rename(species = Species, sepal_length = Sepal.Length,
           sepal_width = Sepal.Width, petal_length = Petal.Width,
           petal_width = Petal.Length)  # classic mistake ;)
## this is the magic line!
usethis::use_data(useful_data, overwrite = TRUE)
```

Run the script! What happened?

See why this convention is helpful? Any person familiar with the R-package workflow expects that an object part of a package was generated in the `data-raw` *folder, in a script with the same name.*

Running the script (see also `?usethis::use_data`) will generate the `data` folder and the R-data `useful_data.rda`.

`devtools::load_all()`

You will run `devtools::load_all()` a lot! It essentially simulates a `library(Rpackage)` call (i.e., loads the package as is it was installed).

```
devtools::load_all()
head(Rpackage::useful_data)
```

```
  sepal_length sepal_width petal_width petal_length species
1          5.1         3.5         1.4          0.2  setosa
2          4.9         3.0         1.4          0.2  setosa
3          4.7         3.2         1.3          0.2  setosa
```

```
4          4.6         3.1         1.5          0.2  setosa
5          5.0         3.6         1.4          0.2  setosa
6          5.4         3.9         1.7          0.4  setosa
```

The `R` *directory*

Everything in the `R` folder gets sourced! Maybe we have a function that we use in multiple scripts (during data preparation or analysis in `data-raw`) or want other users to benefit from (when we share the package)...

It is convention to name the source file the same as the function name. I.e., a function `foo()` *lives in* `foo.R`*! It's just a convention, you can put multiple functions into the same source file if you like...*

```
#= R/foo.R
foo <- function(msg = "Are you sleeping?") {
    cat(msg, "\n")
  }
```

Again, "test it"

By the way, you can put other objects (other than functions) in a script in the `R` *folder... Remember, the scripts get just sourced during package loading (try it out and put* `print("I'm getting sourced")` *in a separate file and simulate loading via* `devtools::load_all()`*).*

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

```
Are you sleeping?
```

What did we learn so far?

- What an (R) package is and (hopefully) we are all convinced that it is pretty easy and useful!
- How to organize raw data and generate analysis-ready data.
- How to "outsource" functions that we frequently use or want other users to be able to consume

Sometimes it is also reasonable to write a function only for organizational sake and not clutter the source file. Usually if you write a function such as `make_tea()` *you can forget about the implementation details...*

3 THE ANALYSIS SCRIPT

As already mentioned, we are mostly writing analysis scripts that only get run "once" and produce some intermediate or final results (either used in another script or in the final paper). Again, there are many ways to do this, but my way is to use the exact same workflow as for raw data. Let's be concrete

```
usethis::use_data_raw("analysis_script")
```

Then put the following code in there, i.e., the file `./data-raw/analysis_script.R` (make sure you understand it!)

```
#= data-raw/analysis_script.R
## code to prepare `analysis_script` dataset goes here
## load the package (as we use the useful_data)
devtools::load_all()
## very short analysis (your analysis is probably more insightful...)
fit <- lm(sepal_length ~ species, data = useful_data)
## and save some useful intermediate results
analysis_script <- list()
analysis_script$fit <- fit
## this is the magic line!
usethis::use_data(analysis_script, overwrite = TRUE)
```

and execute!

Exactly as before, this will generate `analysis_script.rda` in the `data` folder. The `analysis_script` is just a 'list' object containing the `lm` fit, which is now readily available!

```
devtools::load_all()
summary(Rpackage::analysis_script$fit)
```

```
Call:
lm(formula = sepal_length ~ species, data = useful_data)

Residuals:
    Min      1Q  Median      3Q     Max
-1.6880 -0.3285 -0.0060  0.3120  1.3120

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)         5.0060     0.0728  68.762  < 2e-16 ***
speciesversicolor   0.9300     0.1030   9.033 8.77e-16 ***
speciesvirginica    1.5820     0.1030  15.366  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5148 on 147 degrees of freedom
Multiple R-squared:  0.6187,        Adjusted R-squared:  0.6135
F-statistic: 119.3 on 2 and 147 DF,  p-value: < 2.2e-16
```

4 THE DOCUMENTATION GAME

Documentation as "code"

The above gets you already quite far. Did you ever wonder, how these fancy R-helppages are generated? Documentation and code should live together (if possible). We can use the **roxygen2** package to easily document code using a special comment syntax (starting with `#'`). Move back to the `R/foo.R` file, place the cursor inside the function body and press `Shift+Ctrl/Cmd+P`. Type "Insert a rox" and hit enter. This scaffolds the *roxygen skeleton*.

Anything you can achieve in RStudio is available through this command palette (as in most IDEs)...

Let's explain what this function does

```
#= R/foo.R
#' For Orientation Only
#'
#' This function does nothing useful, but it could!
#'
#' @param msg a message to `cat` to the console.
#'
#' @returns NULL
#' @export
#'
#' @examples
#' foo("No, I am not!")
foo <- function(msg = "Are you sleeping?") {
    cat(msg, "\n")
  }
```

`devtools::document()`

This *parses* the roxygen skeleton and generates the `man` folder (manpages are the helppages) as well as the `.Rd` documentation (which is R's markup, similar to LaTeX, for documentation). Further, it automatically manages the `NAMESPACE` for us. But we don't have to know about this – all we need to do, is, write the magic comments and run `devtools::document()` to update it. Check this out

A `NAMESPACE` *defines, what objects are visible when loading the package...*

```
?foo
```

Wicked!

How to document data?

While you are at it, take a look at the `DESCRIPTION` file – it just contains some metadata, minimally documenting the package. Just fill in the fields.

roxygen2 only generates documentation from the source in the `R` directory (at least per default). I usually create a file `R/roxygen.R` where I put all the documentation stuff, with no corresponding source file in the `R` directory, such as our `useful_data` data (remember, its source lives in `data-raw`). So, let's document it

```
#= R/roxygen.R
#' Edgar Anderson's Iris Data
#'
#' With cleaner names...
#'
#' @format Data frame
#' \describe{
#' \item{sepal_width}{Width of the sepal (numeric)}
#' \item{sepal_length}{Length of the sepal (numeric)}
#' \item{petal_width}{Width of the petal (numeric)}
#' \item{petal_length}{Length of the petal (numeric)}
#' \item{species}{Species (factor with levels setosa, versicolor, virginica)}
#' }
"useful_data"
```

Again, run

```
devtools::document()
?useful_data
```

to generate the actual man pages in the `man` directory, and load them.

`devtools::build_manual()` *generates a* `.pdf` *manual for free – try it out!*

5 PAPER VIGNETTE

What is reproducible research? Anyone should be able to generate your results on the fly! Now imagine that you want to present the results of your regression analysis which was conducted in `data-raw/analysis_script.R` and is available via `Rpackage::analysis_script$fit`. You maybe copy paste the parameter estimates to an excel file or use some tool to build a LaTeX table. In my view, this introduces three problems

1. The link between the final output (the table) and the analysis breaks (if you share the LaTeX source, it is not evident, where the results were generated).

2. If you update the analysis (e.g., add a variable in the regression) a whole chain of manual updates is required (which is tidious and error prone)!

3. Building tables manually is not fun... ;)

Enters:	The package vignette
Leaving:	Overleaf

But how do I collaborate on a paper? You might wonder... I would recommend that you only collaborate with people familiar to this workflow (and git)! ;)

`paperPackage::scaffold()` https://github.com/dheimgartner/paperPackage/

R has built-in support to compile `.Rnw` documents – nothing else than plain LaTeX intermingled with R code chunks. Code chunks start with `<<>>=` and end with `@`. Everything in between is evaluated as R-code and its output can be embedded in the text. As a starting point, run

```
paperPackage::scaffold("RJournal")
```

which scaffolds (I like that word!) the `vignettes` folder and generates a minimal article using the *R Journal* style...

`devtools::build_vignettes()`

Compile the vignette using

```
devtools::build_vignettes()
```

which builds the vignette (our paper) in the `doc` repository and will be shipped together with our package (run `vignette("article",package = "Rpackage")`).

You can also use **knitr** *and* `.Rmd` *files to generate vignettes or papers, but I am quite old-fashioned and prefer* `Sweave` *(i.e.,* `.Rnw`*)* https://stat.ethz.ch/R-manual/R-devel/library/utils/doc/Sweave.pdf.

You can also upload the code to GitHub and then people can install it via `devtools::github_install()`. *See* https://github.com/dheimgartner/Rpackage

6 SHARING IS CARING

`devtools::build()`

This builds the source package (`.tar.gz` – a compression file format similar to `.zip`) which can be installed by ...

```
install.packages("../Rpackage.tar.gz", repos = NULL)
```

7 TAKE AWAY

Let's recap the whole game by recalling what these functions do (do you remember?

- `usethis::create_package()`
- `usethis::use_data_raw()`
- `devtools::document()`
- `paperPackage::scaffold()`
- `devtools::build_vignettes()`
- `devtools::load_all()`
- `devtools::build()`

By default, the built package is located in the parent. That's why we need to navigate one folder level up when installing (i.e., `../`*).*

usethis *and* **devtools** *provide many more helpful functions for package development!*

8 NEXT STEPS

- Install this package and explore its "functionality": Inspect the data, call the function, browse the documentation and vignette – make sure you understand, which source files generate the functionality!

- Learning by doing – try it for your next project/paper!
- This gets you going but it only scratches the surface (e.g., how to properly manage external dependencies)
- Tip: Run `devtools::check()` and learn from the errors, warnings and notes...
- Please, please, please learn some git for versioning and your own sake (and necessarily some bash)

Resources

- Package development: https://r-pkgs.org/ (Wickham and Bryan, 2023)
- bash: https://linuxcommand.org/lc3_learning_the_shell.php (Shotts, 2019)
- Git BASH for windows users: https://gitforwindows.org/
- git: https://git-scm.com/book/en/v2 – first three chapters (Chacon and Straub, 2014)

GitHub is "just" a host for repositories – a website/app, but with some very nice features for project management, e.g., the issue tracker). However, the main versioning tool (and magic) is **git** *– don't mistake the two...*

REFERENCES

Chacon S, Straub B (2014). *Pro Git*. 2nd edition. Free e-book: https://git-scm.com/book/en/v2. 7

Shotts W (2019). *The Linux Command Line*. 2nd edition. Turnaround, London. 7

Wickham H, Bryan J (2023). *R Packages (2e)*. 2nd edition. O'Reilly Media, Sebastopol. Free e-book: https://r-pkgs.org/. 7