R is a great resource for data management, statistics, analysis, and visualization — and it becomes better every day. This is to a large part because of the active community that continuously creates and builds extensions for the R world. If you want to contribute to this community, writing a package can be one way. That is exactly what we intended with our package [overviewR](https://cosimameyer.github.io/overviewR/). While there exist many great resources for learning how to write a package in R, we found it difficult to find one all-encompassing guide that is also easily accessible for beginners. This tutorial seeks to close this gap: we will provide you with a step-by-step guide — seasoned with new and helpful packages

In the following sections, we will use a simplified version of one function (overview\_tab) from our overviewR package as a minimal working example.

**Overview**

1. [**Why you should write a package**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#section1)
2. [**Where to start**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#section2)
   1. [Idea](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection2-1)
   2. [Name](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection2-2)
3. [**Set up your package with RStudio and GitHub**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#section3)
4. [**Fill your package with life**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#section4)
   1. [Add functions](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection4-1)
   2. [Write a help file](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection4-2)
   3. [Write DESCRIPTION](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection4-3)
   4. [Add internal data set](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection4-4)
   5. [Write the NEWS.md](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection4-5)
   6. [Write the vignette](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection4-6)
5. [**Check your package**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#section5)
   1. [Write tests](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection5-1)
   2. [codecov](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection5-2)
   3. [Check whether it works on various operating systems with devtools and rhub](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection5-3)
   4. [Generate cran-comments.md file](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection6-1)
   5. [Continuous integration with Travis CI](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection5-4)
   6. [Checking for good practice I: goodpractice](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection5-5)
   7. [Checking for good practice II: inteRgrate](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection5-6)
6. [**Submit to CRAN**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#section6)
7. [**Add-ons**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#section7)
   1. [Create your own hexagon sticker](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection7-1)
   2. [Add badges](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection7-2)
   3. [Create your own manual](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection7-3)
   4. [Build your website for your package](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#subsection7-4)
8. [**Further readings**](https://www.mzes.uni-mannheim.de/socialsciencedatalab/article/r-package/#furtherreadings)

**Why you should write a package**

Writing a package has two main advantages. First, it helps you to approach your problems in a functional way, e.g., by turning your everyday tasks into little functions and bundling them together. Second, it is easy to share your code and new functions with others and thereby contribute to the engaged and vivid R community.

When it comes to our package, we wanted to add an automated way to get an overview — hence the name — of the data you are working with and present it in a neat and accessible way. In particular, our main motivation came from the need to get an overview of the time and scope conditions (i.e., the units of observations and the time span that occur in the data) as this is a recurring issue both in academic articles and real-world situations. While there are ways to semi-automatically extract this information, we were missing an all-integrated function to do this. This is why we started working on overviewR.

To make your package easily accessible for everyone, there are two basic strategies. You can either publish your package on GitHub (which, in terms of transparency, is always a good idea) or you can submit it to the Comprehensive R Archive Network (CRAN). Both offer the ability for others to use your package but differ in several important aspects. Releasing on CRAN offers additional tests that ensure that your package is stable across multiple operating systems and is easily installable with the function utils::install.packages() in R. If you have your package only on GitHub, there is also a function that allows users to install it directly – devtools::install\_github from the devtools package – but most users are more likely to prefer the framework and stability that they can expect from a package that is on CRAN.

We will walk you through both options and start with how to make your package accessible on GitHub before discussing what needs to be done and considered when submitting it to CRAN. To set up your package in RStudio, you need to load the following packages:

library(roxygen2) # In-Line Documentation for R

library(devtools) # Tools to Make Developing R Packages Easier

library(testthat) # Unit Testing for R

library(usethis) # Automate Package and Project Setup

When preparing this post, we came across this incredibly helpful cheat sheet that gives a detailed overview of what the devtools package can do to help you build your own package.

**Where to start**

**Idea**

All good things have to start somewhere and this is most often when you realize that the world is lacking something that is necessary and where you believe others will also benefit from. R packages come in various shapes — from entire universes such as the tidyverse package family (if you look for some Stata like feedback when using the tidyverse and additions to these universes, tidylog is your best friend!), packages for specific statistical models and their validation (icr, MNLpred or oolong), to packages such as polite that offers a netiquette when scraping the web, snakecase that converts names to snake case format, rwhatsapp for scraping WhatsApp, or meme, a package that allows you to make customized memes. As you can tell, the world – and your fantasy – is your oyster.[^1]

**Name**

Let us assume you have a great idea for a new package, the next step would be to find and to pick a proper name for it. As a general rule, package names can only be letters and numbers and must start with a letter. The package [available](https://cran.r-project.org/web/packages/available/index.html) helps you — both with getting inspiration for a name and with checking whether your name is available. This is exactly what we did in our case:

library(available) # Check if the Title of a Package is Available,

# Appropriate and Interesting

# Check for potential names

available::suggest("Easily extract information about your sample")

## easilyr

suggest takes a string with words that can be a description of your package and suggests a name based on this string. As you can tell, we did not go with the suggestion but opted for overviewR instead. We then checked with available whether the name is still available and valid across different platforms. Since our package is already published, it is not available on GitHub, CRAN, or Bioconductor (hence, the “x”).

# Check whether it's available

available::available("overviewR", browse = FALSE)

## -- overviewR ------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Name valid:

## Available on CRAN:

## Available on Bioconductor:

## Available on GitHub:

## Abbreviations: <http://www.abbreviations.com/overview>

## Wikipedia: <https://en.wikipedia.org/wiki/overview>

## Wiktionary: <https://en.wiktionary.org/wiki/overview>

## Urban Dictionary:

## a general [summary] of a subject "the [treasurer] gave [a brief] overview of the financial consequences"

## <http://overview.urbanup.com/3904264>

## Sentiment:???

Let your creativity spark and learn from fantastic package names such as [GeneTonic](https://federicomarini.github.io/GeneTonic/index.html) or [charlatan](https://cran.r-project.org/web/packages/charlatan/index.html).

**Set up your package with RStudio and GitHub**

When setting up your package, there are various possible ways. Ours was to use RStudio and GitHub. RStudio already has a template that comes with the main documents that are necessary to build your package. To access the template, just click on File > New Project... > New Directory > R Package. Note, you need to check the box Create a git to set up a local git.

Hooray, you have started your own package! Let us take a look at the different files that were created.

* .gitignore and .Rbuildignore contain documents that should be ignored when either building in git or R
* DESCRIPTION gives the user all the core information about your package – we will talk more about this below.
* man contains all manuals for your functions. You do not need to touch the .Rd files in there as they will be generated automatically once we populate our package with functions and run [devtools::document()](https://www.rdocumentation.org/packages/devtools/versions/2.3.0/topics/document?tap_a=5644-dce66f&tap_s=10907-287229).
* NAMESPACE will later contain information on exported and imported functions. This file will not be modified by hand but we will show you how to do it automatically. This might seem counter-intuitive in the workflow, but we need to delete the NAMESPACE file here. We do this because we want NAMESPACE to be generated and to be accessible with the devtools universe. We will generate it automatically again later using the command [devtools::document()](https://www.rdocumentation.org/packages/devtools/versions/2.3.0/topics/document?tap_a=5644-dce66f&tap_s=10907-287229).
* R contains all the functions that you create. We will address this folder and its files in the next step.

However, your package is not yet linked with your GitHub. We will do this in the next step:

1. Log in to your [GitHub account](https://github.com/).
2. Create a new repository with “+New Repository”. We named it “overviewR” (as our package). You can set it to private or public – whatever is best for you.
3. Do not check the box “Initialize this Repository with a README”
4. Once you created the repository, execute the following commands in your [RStudio terminal](https://support.rstudio.com/hc/en-us/articles/115010737148-Using-the-RStudio-Terminal):

git remote add origin <https://github.com/YOUR_USERNAME/REPOSITORY_NAME.git>

git commit -m "initial commit"

git commit -u push origin master

If you now refresh your GitHub repository, you will see that your R package is perfectly synchronized with GitHub.

GitHub will now also ask you whether you want to create a README – just click on it and you are ready to go. To get the README in your project, pull it from GitHub either using the Pull button in the Git tab in RStudio or execute the following command line in the RStudio terminal:

git pull

**Fill your package with life**

We will showcase a typical workflow for creating a package using one example function (overview\_tab) from our overviewR package. In practice, you can add as many functions as you want to your package.

**Add functions**

The folder R contains all your functions and each function is saved in a new R file where the function name and the file name are the same. As you can see, the template comes with the preset function hello that returns "Hello, world!" when executed. (The file hello.R showcases the function and can later be deleted.) To include now our function as well, we open a new R file and insert a basic version of our function.

Since we program our function using the tidyverse, we have to take care of the tidy evaluation and use [enquo()](https://dplyr.tidyverse.org/reference/tidyeval.html) for all our inputs that we later modify. Going into detail on how to program in the tidyverse and how and when we need to use enquo, is beyond the scope of this blogpost.

In the preamble of this file, we can add information on the function. An example is shown below:

#' @title overview\_tab

#'

#' @description Provides an overview table for the time and scope conditions of

#' a data set

#'

#' @param dat A data set object

#' @param id Scope (e.g., country codes or individual IDs)

#' @param time Time (e.g., time periods are given by years, months, ...)

#'

#' @return A data frame object that contains a summary of a sample that

#' can later be converted to a TeX output using \code{overview\_print}

#' @examples

#' data(toydata)

#' output\_table <- overview\_tab(dat = toydata, id = ccode, time = year)

#' @export

#' @importFrom dplyr "%>%"

* @title takes the name of your function
* @description contains a short description
* @param takes all your arguments that are in the input of the function with a short description. Our function has three arguments (dat (the data set), id (the scope), and time (the time period)).
* @return gives the user information about the output of your function
* @examples eventually provide a minimal working example for the user to see what s/he needs to include. You can also wrap \dontrun{} around your examples if they should not be executed (e.g., if additional software or an API key is missing). If this is not the case, it is not recommended to wrap this around as it will cause a warning for the user. If your example runs longer than 5 seconds, you can wrap \donttest{} around it.
* @export – if this is a new package, it is always recommended to export your functions. It automatically adds these functions to the NAMESPACE file.
* @importFrom dplyr "%>%" pre-defines required functions for your function. It automatically adds these functions to the NAMESPACE file.

Once you have included the preamble, you can now add your function below.

**Write a help file**

When you execute [devtools::document()](https://www.rdocumentation.org/packages/devtools/versions/2.3.0/topics/document?tap_a=5644-dce66f&tap_s=10907-287229), R automatically generates the respective help file in man as well as the new NAMESPACE file. If you click on it, you see that it is read-only and all edits should be done in the main R function file in R/.

Now you can call the function with ? overview\_tab and get the nice package help that you know from other functions as well.

**Write DESCRIPTION**

The DESCRIPTION is pre-generated by [roxygen2](https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html) and contains all the information about your package that is necessary. We will walk you through the most essential parts:

Type: Package

Package: overviewR

Title: Easily Extracting Information About Your Data

Version: 0.0.2

Authors@R: c(

person("Cosima", "Meyer", email = "XX@XX.com", role = c("cre","aut")),

person("Dennis", "Hammerschmidt", email = "XX@XX.com", role = "aut"))

Description: Makes it easy to display descriptive information on

a data set. Getting an easy overview of a data set by displaying and

visualizing sample information in different tables (e.g., time and

scope conditions). The package also provides publishable TeX code to

present the sample information.

License: GPL-3

Depends:

R (>= 3.5.0)

Imports:

dplyr (>= 1.0.0)

Suggests:

covr,

knitr,

rmarkdown,

spelling,

testthat

VignetteBuilder:

knitr

Encoding: UTF-8

Language: en-US

LazyData: true

RoxygenNote: 7.1.0

* Type: Package should remain unchanged
* Package has your package’s name
* Title is a really short description of your package
* Version has the version number
* Authors@R contains the authors’ names and their roles. [cre] stands for the creator and this person is also the maintainer while [aut] is the author. There are also options to indicate a contributor ([ctb]) or translator ([trl]). At this point, you also need to give your e-mail address. If you want to submit your package to CRAN (but also in any other case), make sure that your e-mail address is correct and accessible!
* Description provides a longer description of what your package does. If you want to indent, use four blank spaces.
* License shows others what they can do with your package. This is an important part and probably a tough decision.
* URL indicates where the package is currently hosted
* BugReports show where users should address their reports to (if linked with GitHub, this will automatically refer the user to the issues section)
* Depends shows the R version your package works with (you always need to indicate a version number!)
* Imports shows the packages that are required to run your package (here you always need to indicate a version number so that potential conflicts with previous versions can be avoided!)
* Suggests lists all the packages that you suggest but that are not necessarily required for the functionality of your package
* …
* LazyData: true ensures that internal data sets are automatically loaded when loading the package

**Add internal data set**

As you know, every good data set, even if it is only a toy data set, comes with a description. For your package, just set up an .R file with the name of your data set (toydata.R in our case) and save it in the R folder. The file should contain the following information:

* Starts with a title for the data set
* Then you have some lines for a short concise description
* @docType defines the type of document (data)
* @usage describes here how the data set should be loaded.
* @format gives information on the object’s format
* \describe{} then allows you to give the user a specific description of your variables included in the data set
* @references is essential if you do not use artificially generated data to indicate the source
* @keywords allows you to indicate keywords (we used dataset here)
* @examples finally gives you some room to showcase your data

What we included in our toydata.R file (you can simply copy and paste the code and adjust it to your needs)

#' Cross-sectional data for countries

#'

#' Small, artificially generated toy data set that comes in a cross-sectional

#' format where the unit of analysis is either country-year or

#' country-year-month. It provides artificial information for five countries

#' (Angola, Benin, France, Rwanda, and the UK) for a time span from 1990 to 1999 to

#' illustrate the use of the package.

#'

#' @docType data

#'

#' @usage data(toydata)

#'

#' @format An object of class \code{"data.frame"}

#' \describe{

#' \item{ccode}{ISO3 country code (as character) for the countries in the

#' sample (Angola, Benin, France, Rwanda, and UK)}

#' \item{year}{A value between 1990 and 1999}

#' \item{month}{An abbreviation (MMM) for month (character)}

#' \item{gpd}{A fake value for GDP (randomly generated)}

#' \item{population}{A fake value for population (randomly generated)}

#' }

#' @references This data set was artificially created for the overviewR package.

#' @keywords datasets

#' @examples

#'

#' data(toydata)

#' head(toydata)

#'

"toydata"

**Write the NEWS.md**

You can automatically generate a NEWS.md file using R with [usethis::use\_news\_md](https://rdrr.io/cran/usethis/man/use_news_md.html). Our news file looks like this:

# overviewR 0.0.2

- Bug fixes in overview\_tab that affected overview\_crosstab

---

# overviewR 0.0.1

The newest release always comes first and --- dividers separate the versions. To inform users, use bullet points to describe changes that came with the new version. As a plus, if you plan to generate a website with [pkgdown](https://pkgdown.r-lib.org/) (we will explain later how you can do this), the news section automatically integrates this file.

**Write the vignette**

A vignette can come in handy and allows you to present the functions of your package in a more elaborate way that is easily accessible for the user. Similar to the news section, your vignette will also be automatically integrated into your website if you use [pkgdown](https://pkgdown.r-lib.org/). You can think of a vignette as something like a blog post that outlines specific use cases or more detailed descriptions of your package.

Here, [usethis](https://usethis.r-lib.org/index.html) offers an excellent service and allows you to create your first vignette automatically with the command [usethis::use\_vignette("NAME-OF-VIGNETTE")](https://usethis.r-lib.org/reference/use_vignette.html). This command does three different things:

1. It generates your vignettes/ folder,
2. Adds essential specifications to the DESCRIPTION, and
3. It also stores a draft vignette “NAME-OF-VIGNETTE.Rmd” in the vignettes folder that you can now access and edit. This draft vignette already contains a nice template that offers you all information and pre-requirements that you need to generate your good-looking vignette. You can adjust this as needed to show what your package does and how it can be used best.

**Check your package**

The following steps are either recommended or required when submitting your package to CRAN. We, however, recommend following all of them. We summarized what we believe is helpful when testing your package.

**Write tests**

Writing tests felt like the most difficult part of building the package. Essentially, you have to come up with tests for every part of your function to make sure that everything – not only the final output of your function – runs smoothly. A piece of good advice, that we read multiple times, is that whenever you encounter a bug, write a test for it to check for future occurrences. To set up the test environment, we used a combination of the great [testthat](https://testthat.r-lib.org/) package and [covr](https://cran.r-project.org/web/packages/covr/index.html) that allows you to visually see how good your test coverage is and which parts of the package still need to be tested.

1. Generate the test environment [usethis::use\_testthat](https://www.rdocumentation.org/packages/usethis/versions/1.5.1/topics/use_testthat?tap_a=5644-dce66f&tap_s=10907-287229). This generates a tests/ folder with another folder called testthat/ that later contains your tests as well as an R file testthat.R. We will only add tests to the tests/testthat/ folder and do not touch the R file.
2. Add test(s) as .R files. The filename does not matter, just choose whatever you find reasonable.
3. Run the tests using [devtools::test()](https://www.rdocumentation.org/packages/devtools/versions/2.3.0/topics/test?tap_a=5644-dce66f&tap_s=10907-287229). To get an estimation of your test coverage, you can use [devtools::test\_coverage()](https://www.rdocumentation.org/packages/devtools/versions/2.3.0/topics/test?tap_a=5644-dce66f&tap_s=10907-287229).

We attach the code that we used to test our overview\_tab() function below and hope this sparks some inspiration when testing your functions.

Code for function testing

context("check-output") # Our file is called "test-check\_output.R"

library(testthat) # load testthat package

library(overviewR) # load our package

# Test whether the output is a data frame

test\_that("overview\_tab() returns a data frame", {

output\_table <- overview\_tab(dat = toydata, id = ccode, time = year)

expect\_is(output\_table, "data.frame")

})

# In reality, our function is more complex and aggregates your input if you have duplicates in your id-time units -- this is why the following two tests were essential for us

## Test whether the output contains the right number of rows

test\_that("overview\_tab() returns a dataframe with correct number of rows", {

output\_table <- overview\_tab(dat = toydata, id = ccode, time = year)

expect\_equal(nrow(output\_table), length(unique(toydata$ccode)))

})

## Test whether the function works on a data frame that has no duplicates in id-time

test\_that("overview\_tab() works on a dataframe that is already in the correct

format",

{

df\_com <- data.frame(

# Countries

ccode = c(

rep("RWA", 4),

rep("AGO", 8),

rep("BEN", 2),

rep("GBR", 5),

rep("FRA", 3)

),

# Time frame

year =

c(

seq(1990, 1995),

seq(1990, 1992),

seq(1995, 1999),

seq(1991, 1999, by = 2),

seq(1993, 1999, by = 3)

)

)

output\_table <-

overview\_tab(dat = df\_com, id = ccode, time = year)

expect\_equal(nrow(output\_table), 5)

})

**codecov**

Once you are done with your tests, you can also link your results automatically with [codecov.io](https://codecov.io/login) to your GitHub repository. This allows codecov to automatically check your tests after each push to the repository. As a bonus, you will also get a nice badge that you can then be included in your GitHub README to show the percentage of passed tests for your package.

To link codecov and GitHub, simply follow these steps:

1. Log in on [codecov.io](https://codecov.io/login) with your GitHub account
2. Give codecov access to your repository with your package
3. This will prompt a page where you can copy your token from
4. Now go back to your RStudio console and execute:

library(covr) # Test Coverage for Packages

covr::codecov(token = "INCLUDE\_YOUR\_CODECOV\_TOKEN\_HERE")

1. This will then link your GitHub repository with codecov and generate the badge.

**Check whether it works on various operating systems with devtools and rhub**

To check whether our package works on various operating systems, we relied on a combination of the [rhub](https://cran.r-project.org/web/packages/rhub/vignettes/rhub.html) and [devtools](https://devtools.r-lib.org/) packages. These packages also help you to integrate a continuous integration with Travis CI that automatically checks your package on Ubuntu (more on this below).

We used the following lines of code sequentially to check our package:

# The following function runs a local R CMD check

devtools::check()

This command can take some time and produces an output in the console where you get specific feedback on potential errors, warnings, or notes.

# Check for CRAN specific requirements

rhub::check\_for\_cran()

This command checks for standard requirements as specified by CRAN and, if saved in an object, you can generate your cran-comments.md file based on this command. We will go into further detail about this in the next section. If you use [rhub](https://r-hub.github.io/rhub/articles/rhub.html) for the first time, you need to validate your e-mail address with [rhub::validate\_email()](https://www.rdocumentation.org/packages/rhub/versions/1.0.0/topics/validate_email?tap_a=5644-dce66f&tap_s=10907-287229). You can then execute the command. Once the command ran, you will receive three different e-mails that give you detailed feedback on how well the tests performed on three different operating systems. At the time of writing, this function checked our package on Windows Server 2008 R2 SP1, R-devel, 32/64 bit; Ubuntu Linux 16.04 LTS, R-release, GCC; and Fedora Linux, R-devel, clang, gfortran.  
From our experience, the checks on Windows were extremely fast but we had to wait a bit until we got the results for Ubuntu and Fedora.

We then also checked the package on the development version of R as suggested with the following function:

# Check for win-builder

devtools::check\_win\_devel()

**Generate cran-comments.md file**

If you plan to submit your package to CRAN, you should save your test results in a cran-comments.md file. [rhub](https://cran.r-project.org/web/packages/rhub/vignettes/rhub.html) and [usethis](https://usethis.r-lib.org/index.html) allow us to create this file almost automatically using the following lines of code:

# Check for CRAN specific requirements using rhub and save it in the results

# objects

results <- rhub::check\_for\_cran()

# Get the summary of your results

results$cran\_summary()

We received the following output when running the results$cran\_summary() command.

For a CRAN submission we recommend that you fix all NOTEs, WARNINGs and ERRORs.

## Test environments

- R-hub windows-x86\_64-devel (r-devel)

- R-hub ubuntu-gcc-release (r-release)

- R-hub fedora-clang-devel (r-devel)

## R CMD check results

> On windows-x86\_64-devel (r-devel), ubuntu-gcc-release (r-release), fedora-clang-devel (r-devel)

checking CRAN incoming feasibility ... NOTE

New submission

Maintainer: 'Cosima Meyer '

0 errors ✓ | 0 warnings ✓ | 1 note x

Your package must not cause any errors or warnings when submitting to CRAN. Even notes need to be well explained. In our case, we receive one note saying that this is a new submission. This note occurs every time when you submit a new package and can be briefly be explained when submitting your package to CRAN in the cran-comments.md file.

We then generated our cran-comments.md file with the following command and copy-pasted this output with minor adjustments.

# Generate your cran-comments.md, then you copy-paste the output from the function above

usethis::use\_cran\_comments()

**Continuous integration with Travis CI**

Continuous integration (CI) is incredibly helpful to ensure the smooth working of your package every time you update even small parts. Using the command [usethis::use\_travis()](https://usethis.r-lib.org/reference/ci.html) you can easily link Travis CI with your GitHub repository. Travis CI then checks your package after each push to your repository on Ubuntu. Explaining CI in further detail would require another blog post or book itself.

**Checking for good practice I: goodpractice**

The package [goodpractice](https://github.com/mangothecat/goodpractice) is incredibly helpful and provides you all the information that you need when it comes to polishing your package with concerning your syntax, package structure, code complexity, formatting, and much more. And, the best thing: it provides easily understandable feedback that pinpoints you exactly to the lines of code where changes are recommended.

libary(goodpractice)

goodpractice::gp()

As a general tip for improving the style of your code, the package [styler](https://github.com/r-lib/styler) provides an easy solution by formatting your entire source code in adherence to the tidyverse style (similar to RStudio’s built-in hotkey combination with Cmd + Shift + A (Mac) or Ctrl + Shift + A (Windows)).

**Checking for good practice II: inteRgrate**

The underlying idea behind this package is that it tests stricter than other packages with clear standards. By this, it aims to ensure that you are definitely on the safe side when submitting your package to CRAN. A good starting point is this, we particularly highlight the following functions:

* check\_pkg() installs package dependencies, builds, and installs the package, before running package check (by default this check is rather strict and any note or warning raises an error by default)
* check\_lintr() runs [lintr](https://github.com/jimhester/lintr) on the package, README, and the vignette. lintr checks whether your code adheres to certain standards and that you avoid syntax errors and semantic issues.
* check\_tidy\_description() makes sure that your DESCRIPTION file is tidy. If not, you can use [usethis::use\_tidy\_description()](https://usethis.r-lib.org/reference/tidyverse.html) to follows the tidyverse conventions for formatting.
* check\_r\_filenames() checks that all file extensions are .R and all names are lower case.
* check\_gitignore() checks whether .gitignore contains standard files.
* check\_version() ensures that you update your package version (might be good to run as the last step)

**Submit to CRAN**

Submitting a package to CRAN is substantially more work than making it available on GitHub. It, however, forces you to test your package on various operating systems and ensures that it is stable across all these systems. In the end, your package will become more user-friendly and accessible for a larger share of users. After going through the entire process, we believe that it is worth the effort – just for these simple reasons alone. When testing our package for CRAN. The column *Needed* is based on what is asked for when running [devtools::release()](http://r-pkgs.had.co.nz/release.html). *Recommended* includes additional neat checks that we found helpful.

| **Checks** | **Needed** | **Recommended** |
| --- | --- | --- |
| Update your R, Rstudio and all dependent R packages (R and Rstudio has to be updated manually, devtools::install\_deps() updates the dependencies for you) |  | x |
| Write tests and check if your own tests work (devtools::test() and devtools::test\_coverage() to see how much of your package is covered by your tests) |  | x |
| Check your examples in your manuals (devtools::run\_examples(); unless you set your examples to \dontrun{} or \donttest{}) |  | x |
| Local R CMD check (devtools::check()) | x | x |
| Use devtools and rhub to check for CRAN specific requirements (rhub::check\_for\_cran() and/or devtools::check\_rhub() – remember, you can store your output of these functions and generate your cran-comments.md automatically) | x | x |
| Check win-builder (devtools::check\_win\_devel()) | x | x |
| Check with Travis CI (usethis::use\_travis()) |  | x |
| Update your manuals (devtools::document()) | x | x |
| Update your NEWS file | x | x |
| Update DESCRIPTION (e.g. version number) | x | x |
| Spell check (devtools::spell\_check()) | x | x |
| Run goodpractice check (goodpractice::gp()) |  | x |
| Check package dependencies (inteRgrate::check\_pkg()) |  | x |
| Check if code adheres to standards (inteRgrate::check\_lintr()) |  | x |
| Check if your description is tidy (inteRgrate::check\_tidy\_description() – if your description is not tidy, it will produce an error and ask you to run usethis::use\_tidy\_description() to make your DESCRIPTION tidy) |  | x |
| Check if file names are correct (inteRgrate::check\_r\_filenames()) |  | x |
| Check if .gitignore contains standard files (inteRgrate::check\_gitignore()) |  | x |
| Update cran-comments.md | x | x |
| Run devtools::check() one last time |  | x |

CRAN also offers a detailed policy for package submissions as well as a check list when submitting your package. We definitely recommend to check them in addition to our list above.

As already mentioned above, it is of vital importance that your package must not cause any errors or warnings when submitting to CRAN. Even notes need to be well explained. If you submit a new package, there is not much you can do about it and it will always create a note.

The function [devtools::release()](http://r-pkgs.had.co.nz/release.html) allows you to easily submit your package to CRAN – it works like a charm. Once you feel you are ready, make sure to push your changes to GitHub and then just type the command in your console. It runs a couple of yes-no questions before the submission. The following questions are those asked in the [devtools::release()](https://github.com/r-lib/devtools/blob/master/R/release.R) function at the date of writing this post.

* Have you checked for spelling errors (with spell\_check())?
* Have you run R CMD check locally?
* Were devtool’s checks successful?
* Have you checked on R-hub (with check\_rhub())?
* Have you checked on win-builder (with check\_win\_devel())?
* Have you updated NEWS.md file?
* Have you updated DESCRIPTION?
* Have you updated cran-comments.md?

Once submitted, you will receive an e-mail that requires you to confirm your submission – and then you will have to wait. If it is a new package, CRAN also runs a couple of additional tests and it might take longer than submitting an updated version of your package.

For us, it took about four days until we heard back from CRAN. We read that CRAN is curated by volunteers that can receive and incredible amount of submissions per day. Our experience was extremely positive and supportive which we truly enjoyed.

Once CRAN gets back to you, they will tell you about potential problems that you have to address before resubmitting your package – or you are lucky and your package gets accepted immediately.

Before resubmitting your package, go through all the steps presented in “Submit to CRAN” once again to make sure that your updated version still adheres to the standards of CRAN.

Common things that we have learned (and that others might find helpful) while going through the CRAN submission process are:

1. Do not modify (save or delete) outputs on the user’s home filespace. Use tempdir() and/or tempfile()instead when running examples/vignettes/tests.
2. Make sure that the user can set the directory and the file name when saving outputs. Simply add a file/path argument to your function(s).
3. Write package names, software names, and API names in single quotes in your DESCRIPTION. If you use for example *LaTeX* in your DESCRIPTION, put it in single quotes. This issue is apparently not discovered by goodpractice::gp() or one of the inteRgrate functions.

Afterwards, go to your GitHub repository, click on “Create a new release”, enter the version number of your package (vX.X.X) and copy-paste the release notes from your NEWS file into the release description.

When submitting your package to CRAN via the devtools::release() function, a CRAN-RELEASE file was generated to remind you to tag your release on GitHub. This file can now safely be deleted.

**Add-ons**

This section on add-ons can be considered a bonus. It is not essential to guarantee that your package works smoothly or gets published on CRAN — but the extensions make your package look nicer, more professional, and might help to get discovered by other users.

**Create your own hexagon sticker**

Hex(agon) stickers are the small hexagon-shaped icons that a large number of packages have and that people seem to love. So why not come up with your own sticker for your very own package? The package [hexSticker](https://github.com/GuangchuangYu/hexSticker) makes it incredibly easy to customize and build a beautiful sticker. To get a sticker for your package, just add the following arguments to the function [hexSticker::sticker()](https://rdrr.io/cran/hexSticker/man/sticker.html): package (the name of your package), subplot (an image – we have drawn our lamp ourselves, saved it as a .png and included it in our sticker without any problems), and h\_fill (if you want to change the background color). You can then adjust the sticker by defining the position of the text, the subplot, the font size, or even add a spotlight as we did.  
This works with virtually any text and image combination – also with the Methods Bites logo.

Code for the overviewR sticker

library(hexSticker) # Create Hexagon Sticker in R

library(showtext) # Using Fonts More Easily in R Graphs

## Loading Google fonts (<http://www.google.com/fonts>)

font\_add\_google("Inconsolata", "incon")

sticker(

# Subplot (image)

subplot = "logo-image.png", # Image name

s\_y = 1, # Position of the sub plot (y)

s\_x = 1.05, # Position of the sub plot (x)

s\_width = 1.15, # Width of the sub plot

s\_height=0.01, # Height of the sub plot

# Font

package = "overviewR", # Package name (will be printed on the sticker)

p\_size = 6, # Font size of the text

p\_y = 0.8, # Position of the font (y)

p\_x=0.75, # Position of the font (x)

p\_family = "incon", # Defines font

# Spotlight

spotlight = TRUE, # Enables spotlight

l\_y=0.8, # Position of spotlight (y)

l\_x=0.7, # Position of spotlight (x)

# Sticker colors

h\_fill = "#5d8aa6", # Color for background

h\_color = "#2A5773", # Color for border

# Resolution

dpi=1200, # Sets DPI

# Save

filename="logo.png" # Sets file name and location where to store the sticker

)

Figures such as your logo are usually stored in man/figures/.

**Add badges**

Badges in your GitHub repository are a bit like stickers but they also serve an informative purpose. We included five different badges in our README in our GitHub repository: a Travis CI status, a codecov status, as well as the repo status. We then also added a badge that signals that the package is ready to use and another one that tells the user that the package was built with R (… and love!).

If you want to learn more about available badges for your package, You can also use the package [badgecreatr](https://cran.r-project.org/web/packages/badgecreatr/badgecreatr.pdf) to check for badges and to include them.

**Create your own manual**

If you want to create your own PDF manual for your package, [devtools::build\_manual](https://devtools.r-lib.org/reference/build_manual.html) does this for you.

A preview to our manual

**Build your website for your package**

As the last part, to advertise your package and to provide a more detailed insight into how your package works, you can set up a whole, stand-alone website for it! The [pkgdown](https://pkgdown.r-lib.org/) package makes this as easy as writing one line of code — literally! All you have to do is to install and load pkgdown and then — provided that you have taken all the steps above and have an R-package-structure in your GitHub repository — run [pkgdown::build\_site()](https://pkgdown.r-lib.org/reference/build_site.html). This automatically renders your package into a website that follows the structure of your package with a landing page based on the README file, a “get started” part of your vignette, as well as sections for function references based on the content of your man/ folder, and a dedicated page for your NEWS.md. It even includes a sidebar with links to the GitHub repository, the name(s) of the author(s), and, of course, all your badges. Amazing, right?

Naturally, pkgdown allows for further modifications of your websites’ appearance such as different themes, modified landing pages, different outlines of your navigation bar, etc.

By default, your website is hosted on GitHub pages with the following URL: <https://GITHUB_USERNAME.github.io/PACKAGENAME>. To ensure that every time you update your package, the website gets updated as well, you can modify the .travis.yml file in the root of your package and add the following parts to it:

after\_success:

- Rscript -e 'pkgdown::build\_site()'

deploy:

provider: pages

skip-cleanup: true

github-token: $GITHUB\_PAT

keep-history: true

local-dir: docs

on:

branch: master

This makes sure that every time you push your updates to GitHub, Travis CI will not only check the build of your package but will also update your website automatically.