R-packages

A soft introduction

Jørn I. Halvorsen

2018/06/1 (updated: 2021-10-13)

Introduction

“Seriously, it doesn’t have to be about sharing your code (although that is an added benefit!). It is about saving yourself time.” Organising code in a package makes your life easier because packages come with conventions.

> Hilary Parker (2014)

Hilary (2020)

Start up knowledge

- Writing-an-r-package-from-scratch (Hilary Parker)

General documentation

- R-packages, Wickham (2015) 1st edition
- R-packages, 2nd edition
- Package development

Some sub-themes in relation to this that we will touch:

Data wrangling

- dplyr

Graphical representation

- ggplot2

Webpage development and presentations

- pgkdown
- xaringan
- xaringanExtra

Outline of today's lecture

- 1. Creating a package
- 2. Publishing the package to a repository
- 3. Overview of the different package components
- 4. Build websites for your R-package

1. Creating a package

Before inititating the package

System setup

```
install.packages(c("devtools", "roxygen2","usethis",
"rstudioapi", "testthat", "knitr", "available", "pgkdown","purrr"))
```

For running this presentation

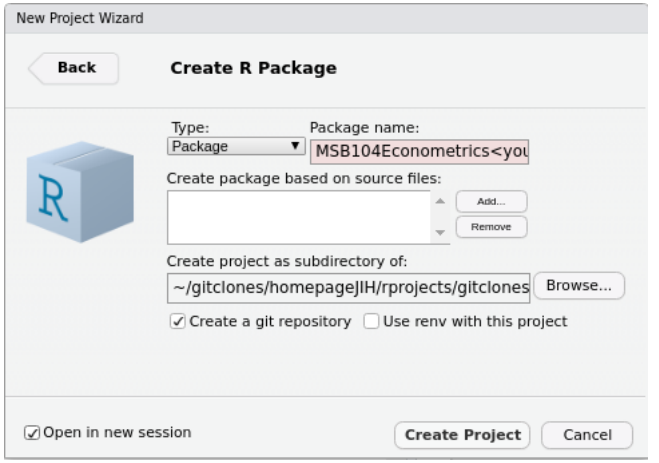
```
install.packages(c("xaringan"))
devtools::install_github("gadenbuie/xaringanExtra")
```

There are three formal requirements:

- The name can only consist of letters, numbers, and period. Must not start with a number
- It must start with a letter
- It cannot end with a period
- Pakkenavn:
    - MSB104TimeSeries<your_init>

RStudio IDE (option A)

File -> New Project -> New Directory -> R package



Console (option B)

```
package_name <- 'MSB104TimeSeries<YourInit>'
usethis::create_package(package_name, rstudio=T, open=T)
```

2. Publishing the package to a repository

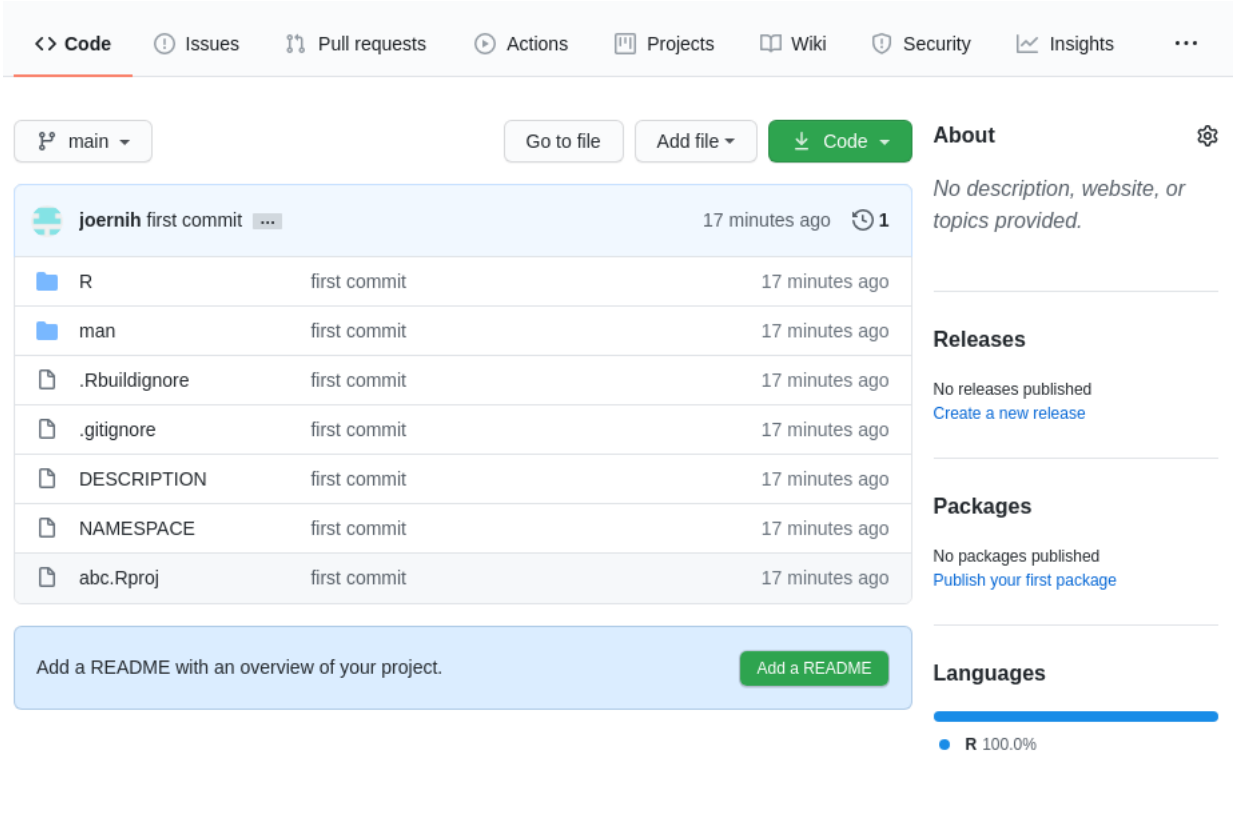
```
git init --initial-branch=main
git add --all
git commit -m "first commit"
```

Now, for either of the options

1. Go to your own github or gitlab account
2. Create a new project (public) with the same name: MSB104TimeSeries<your_init>
3. After that, run the following commands from your RStudio terminal

```
a) Github
git remote add origin https://github.com/<your_github_account>/MSB104TimeSeries<your_init>.git
b) Gitlab
git remote add origin https://gilab.com/<your_github_account>/MSB104TimeSeries<your_init>.git
git branch -M main
git push -u origin main
```

Should look something like this



3. Overview of the different package components

In general consist of:

- Code (R/)
- Package metadata (DESCRIPTION)
- Object documentation (man/)
- Vignettes (vignettes/)
- Testing (tests/)
- Namespaces (NAMESPACE)
- Data (data/)
- Compiled code (src/)
- Installed files (inst/)
- Other components

Note: In addition to this, we have also the option to create our own folders

Let's us now first have a closer look at some of these general components:

- Code (R/)
- Namespaces (NAMESPACE)
- Package metadata (DESCRIPTION)
- Object documentation (man/)
- Data (data/)
- Namespaces (NAMESPACE)

Code (R/)

The first principle of using a package is that all R code goes in R/.

We can open our hello function directly from the console by using

- usethis::use_r("hello")

Which gives us

```
hello <- function() {
          tekst <- paste("Hello world!")
      print(tekst)
}
```

Change this to

```
hello <- function(name=NULL) {
          tekst <- paste("Hello world! My name is", name)
      print(tekst)
}
```

Namespaces (NAMESPACE)

Namespaces make your packages self-contained in two ways: the imports and the exports. The imports defines how a function in one package finds a function in another.

The exports helps you avoid conflicts with other packages by specifying which functions are available outside of your package (internal functions are available only within your package and can’t easily be used by another package).

With RStudio IDE

- Add roxygen comments to your .R files.
    - Roxygen comments start with #' and come before a function. All the roxygen lines preceding a function are called a block. Each line should be wrapped in the same way as your code, normally at 80 characters.
- To make it simple in the beginning, we just add a couple of lines:

```
#' This is a hello function with a name argument
#' @export
hello <- function() {
      print("Hello, world! My name is <insert your name here> ")
}
```

(1) Building (2) installing and (3) loading a package

- Building and installing

2a. Option A (IDE)

```
- Install Package: 'Ctrl + Shift + B'
- MSB104TimeSeries<your_init>::hello()
```

2b. Option B (Console)

1. devtools::document()
2. devtools::install()
3. MSB104TimeSeries<your_init>::hello() A built package can be found in one of these paths

```
.libPaths()
```

```
## [1] "/home/joernih/R/x86_64-pc-linux-gnu-library/4.0"
## [2] "/usr/lib/R/library"
```

- Loading

```
library(MMSB104TimeSeries<your_init>)
### Package consists of the following functions
ls(package:MMSB104TimeSeries<your_init>)
```

Package development (cheat sheet)

Package metadata (DESCRIPTION)

The job of the DESCRIPTION file is to store important metadata about your package

```
Package: MSB104TimeSeriesJIH
Title: For the time series part of the MSB104 course in econometrics
Version: 0.0.0.9000
Authors@R:
    person(given = "Jørn I.",
           family = "Halvorsen",
           role = c("aut", "cre"),
           email = "jiha@hvl.no",
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to pick a
    license
Encoding: UTF-8
LazyData: true
```

Object documentation (man/)

Documentation is one of the most important aspects of a good package.

- Without it users won't know how to use your package.
- Documentation is also useful for future-you (so you remember what your functions were supposed to do)
- For developers extending your package.

Data (data/)

```
usethis::use_data_raw()
file.edit("data-raw/hello_world_data.R")
```

- Change the DATASET.R file to

```
## code to prepare `DATASET` dataset goes here
hw_iris <- iris
usethis::use_data(hw_iris, overwrite = TRUE)
```

- Source the code
- (1) Build, (2) install and (3) load the package
- Run data(package='MSB104TimeSeries')
- You could also try out the data() function

Some other componnents

Testing (tests/)

Testing is a vital part of package development.

- It ensures that your code does what you want it to do.
- Testing, however, adds an additional step to your development workflow.

"I started using automated tests because I discovered I was spending too much time re-fixing bugs that I’d already fixed before."

> Hadley (2015)

Installed files (inst/)

When a package is built, everything in inst/ is copied into the top-level package directory.

You are free to put anything you like in inst/ with one caution: subdirectory with the same name as an existing directory.

- inst/extdata: additional external data for examples and vignettes. See external data for more details
- inst/java,
- inst/python etc. See other languages.

For MSB104TimeSeriesJIH you will find the Rmd code for this xaringan-presentation in

- inst/docs/package_presentationJIH.Rmd

Check if you are able to download this package in comple the presentation on your own computer.

```
### vignettes (vignettes/)

Many existing packages have vignettes.
- You can see all the installed vignettes with


```r
vign <- browseVignettes(all=T)
names(vign)
```

- To see the vignette for a specific package, use the argument

```
browseVignettes("dplyr")
```

- To make your own vignette

```
usethis::use_vignette("hello_world")
```
```

Compiled code (src/)

R is a high-level, expressive language. But that expressivity comes at a price: speed. That's why incorporating a low-level, compiled language like C or C++ can powerfully complement your R code. While C and C++ often require more lines of code (and more careful thought) to solve the same problem, they can be orders of magnitude faster than R.

4. Build websites for your R-package

Configuration and building

```
### Run once to configure your package to use pkgdown
usethis::use_pkgdown()
## Run to build the website
pkgdown::build_site()
```

Deploying the webpage

1. Commit and push your changes to the remote (note: make sure that docs is not included in the .gitignore file)
2. Login on to github or gitlab account, navigate to your repo and select “Settings”.
3. Scroll down to find the “Git Hub pages” subsection and, under “Source”, select “master branch/docs folder” (from this, you can see that it is fundamental that your website material is pushed to the master branch)
4. click on the link of your website and… congratulations: you just published your new pkgdown website !

My deployed github webpage should be deployed here

More material on how to go about and configure your own web page can be found here

-Hadley's description of his own package

-Building a website with pkgdown: a short guide

Application 1: Hello world!

1. Show part of the iris dataset and implement the the hello function in the hello_world vignette
2. Preview the document in the browser from you own local machine
3. Deploy the webpage to the github or gitlab repository

Application 2: Covid-19 time series analysis (task for xx.xx.2021)**

Start first by generating the following vignette

```
usethis::use_vignette("Covid19TimeSeriesAnalysis")
```

1. To the data-raw folder, add the dataset file that you one the next slide
2. To the R-folder, add the function found on the slide after the dataset file
3. In the *Covid19TimeSeriesAnalysis.Rmd*, employ the *plot19ts* function in the Rmakdown document
4. Preview the document in the browser from you own local machine (pkgdown::build_site)
5. Deploy the webpage to the github or gitlab repository

Covid 19-data transformation

```
library(COVID19)
library(dplyr)
library(lubridate)
library(zoo)

all_data <- COVID19::covid19(verbose = F)
unique(all_data$id)

sel_cou <- c('NOR','ITA','SWE','GBR','ISR','FIN','CZE','ESP','USA','CAN','SVK','IND')

COVID19 <- all_data %>% dplyr::filter(id%in%sel_cou) %>%
#### Datering
dplyr::mutate(year=as.factor(lubridate::year(date))) %>%
dplyr::mutate(dayofyear=lubridate::yday(date)) %>%
dplyr::select(id,date,confirmed,deaths,hosp,dayofyear,year,population) %>%
dplyr::mutate(c_deaths=deaths-dplyr::lag(deaths)) %>%
dplyr::mutate(ma_deaths=round(rollmean(c_deaths,k=7, fill=NA)),digits=4) %>%
dplyr::mutate(ma_deaths_perc=(ma_deaths/population)*100000)
usethis::use_data(COVID19, overwrite = TRUE)
```

Covid 19 plot functions

```
#' This is a function for producing Covid-19 time series plot
#' @export
plot19ts <- function(sel_cou=NULL,covid19df=NULL, yvar='hosp'){

    c_cdf <- sel_cou %>% purrr::map(function(x,df=covid_sel_df){
        print(x)
        out <- df %>% dplyr::filter(id==x)
        print(out)
        gout <- ggplot(out, aes(x=dayofyear,y=hosp, color=year)) + geom_point() + geom_line() + labs(title=x,"D
    })
}
```

Covid 19 plot vignette

References

Hilary, P. (2020). *Not So Standard Deviations*. " O'Reilly Media, Inc.".

Wickham, H. (2015). *R packages: organize, test, document, and share your code*. " O'Reilly Media, Inc.".

```
knitr::knit_exit()
```