Package management

Creating and maintaining project-relative package libraries with `renv`

Daniela Palleschi

2024-08-23

Table of contents

Learning objectives

Today we will...

- learn about R package repositories
- learn how package dependencies can break code
- use the `renv` package to create and maintain a project-relative package library

Resources

- to read more on today's topic, check out:
- Ch. 10 (Basic reprodubility: freezing packages) from Rodrigues (2023)
- the `renv` website
  - or CRAN documentation and vignettes therein (e.g.,: Introduction to renv)

Packages

- most open source software (like R) has a range of libraries available
  - created by other users/developers and shared for free
- the benefit of open software (besides being free) is that we don't have to wait for an updated version to be released by a company
  - and *anybody* can create an R package to facilitate certain tasks or fix some problem
- this is part of the reason for the success and popularity of R
  - someone else has likely created a package for some problem or need you have

CRAN packages

- the Comprehensive R Archive Network: R's central software repository
  - currently 20,888 available!
- an archive of the most recent package versions
- for a package to be included in the CRAN, it must go through a lot of tests and checks
  - any updates or changes must again be reviewed before being added to CRAN
- CRAN packages can be installed using `install.packages()`, as we've been doing

`pacman` package (optional)

- a package management tool
- we'll use the `p_load()` function to replace `install.packages()` and `library()` in our worksflow
  - takes a list of packages, and checks if each package is installed already
  - if *yes*, the package is loaded (as with `library()`)
  - if *no*, the package is installed (as with `install.packages()`) and then loaded (as with `library()`)
- only works with CRAN packages (which is all we have for now anyway), although `pacman` has a function for developer packages (which we'll talk about later)

To get started: install `pacman` (`install.packages("pacman")`). Then, you can load in your packages using `pacman::p_load()`, or with a long list of `library()` calls like we've previously done (you see why I prefer `p_load()`!).

Listing 1 Loading packages with `pacman::p_load()`

```
pacman::p_load(tidyverse, here, janitor)
```

Listing 2 Loading packages with `library()`

```
library(tidyverse)
library(here)
library(janitor)
```

The additional benefit of `p_load()` is that, if you don't actually have one of the packages installed it will automatically be installed and then loaded. With `library()` you would instead get an error message.

Developer packages

- often hosted on GitHub or GitLab, where packages are typically developed before being reviewed and added to the CRAN
  - benefit: you can make whatever you changes to your package that you like without having to pass a review on the CRAN
- since CRAN packages are often developed on GH or GL, pre-release (beta) versions will often be available on a GH repo
- packages/package versions on GH cannot be installed via `install.packages()`
  - we'll see later how to do this

Dependencies

- some packages are dependent on specific versions of other packages
  - if so, you will be prompted during installation to install these dependencies
  - but beware: sometimes this overwrites an existing package version you already have, which can break code that was written with this older version
- this is especially true because, as our projects are currently set up, we have one global package version on our computer
  - so analyses we ran 3 years ago would've used older versions of packages
  - when we update these packages for current analyses, this might disrupt the code from 3 years ago
- we'll see one (partial) solution for this problem soon

Package versions

- packages can be updated at any time
  - if hosted on the CRAN, they newer versions are first reviewed/rigorously tested
  - if hosted on GitHub/Lab, nobody needs to check the update before publication
- if you want to check which version of a package you're using, you can run `packageVersion("package")`

```
packageVersion("ggplot2")
```

```
[1] '3.5.1'
```

Updating packages

- to check if a package needs updating, you can:
  - go to `Tools > Check for package updates`, or
  - run `update.packages()`
- each will tell you which packages can be updated to which versions
  - and give you the option of updating these packages

Package library

- where do all these installed packages go?
  - a folder that contains all the packages, called a library
- to find out where this (global) package library is, run `.libPaths()`

```
.libPaths()
```

- the output should currently produce a single file path, something like:

```
> .libPaths()
[1] "/Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/library"
```

- this is the location of your global package library

Package versions and reproducibility

- we've seen that package versions and dependencies can easily break our existing code
- this means that older projects that were built using previous package versions won't be able to run if we update these packages in our global package library
  - also a problem in the future: our current code will depend on the package versions we're using today
- we need a project-relative package library that is independent of the global library
  - we'll use the `renv` package to do this

Listing 3 `Run in the Console`

```
install.packages("renv")
```

`renv`

- `renv` aids in maintaining *r*eproducible *env*ironments in R projects
- available on the CRAN

- main benefit: creates a self-contained, independent library per R Project
  - avoids cross-library package contamination
- `renv` freezes and stores package versions used in a project
- but does not make a project reproducible across R versions and machines
  - that's because older package versions are not always compatible with newer computational environments

Limits of `renv`

`renv`...

...can

- keep track of packages and their versions
- create a project-specific library per R version
- automatically load/install these package versions

...cannot

- make a project reproducible across all computational environments
- load/install package versions that are incompatible with current R versions or computational environments
- guarantee full long-term reproduciblity

`renv` workflow

- Figure 1 visualises a project workflow with `renv`
- next we'll see how we use these functions to set-up and maintain a project-specific package library

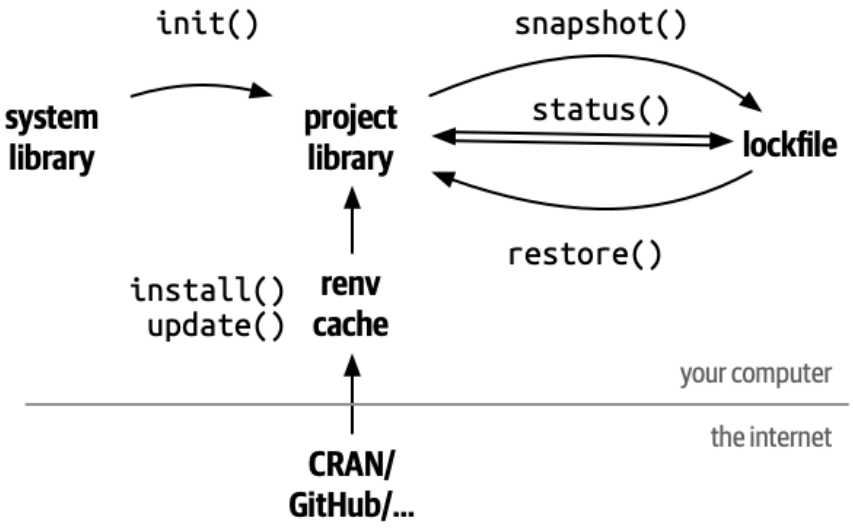

Figure 1: Source: CRAN vignette 'Introduction to renv' (all rights reserved)

Initialise project library

- run the following in the Console *or* in a code chunk but with `#| eval: false`
  - we only want to run this *once* per R Project
  - when working in an actual project, I would just run this in the console
  - for learning/documenting how to use `renv`, I would keep this in a code chunk with `#| eval: false`

Listing 4 `In the Console or with eval: false`

```
renv::init()
```

- you should see something like this in the Console:

```
- Linking packages into the project library ... [137/137] Done!
- Resolving missing dependencies ...
# Installing packages --------------------------------------------------------
The following package(s) will be updated in the lockfile:

# CRAN -----------------------------------------------------------------------
[long list of packages and their versions]
```

```
The version of R recorded in the lockfile will be updated:
- R               [* -> 4.4.0]

- Lockfile written to "~/Documents/IdSL/Teaching/SoSe24/M.A./r4repro_student/renv.lock".

Restarting R session...

- Project '~/Documents/IdSL/Teaching/SoSe24/M.A./r4repro_student' loaded. [renv 1.0.7]
```

New files

- `renv::init()` creates three new files or directories
  - `renv.lock`
  - `renv/`
  - `.Rprofile`
- explore these files/folders and see if you can figure out what they contain

`renv.lock`

- contains metadata about the packages and their versions that you have installed
  - this is enough metadata to re-install these package versions on a new machine
- two main components:
  - `R`: info on R version and list of repositories where packages were installed from
  - `Packages`: a record per package with necessary info for re-installation

`renv/`

- importantly, contains your project-relative `library/`
  - this is instead of using the one library on your computer
- provides us with "isolation": the package versions used in an R Project is independent of the global library
  - in other words, different R Projects can use different package versions
  - updating packages globally, or in one project, will not affect other project libraries

`.RProfile`

- runs whenver you (re-)start your R Project
- at this point, should contain a single line:

```
source("renv/activate.R")
```

- if you go to this R script, you'll send a lot of code
  - this essentially loads in your project library

Project library

- now if we re-run `.libPaths()`, we should see our project library

Listing 5 `Run in the Console`

```
.libPaths()
```

```
> .libPaths()
[1] "/Users/danielapalleschi/Documents/IdSL/Teaching/SoSe24/M.A./r4repro_SoSe2024/renv/libra
[2] "/Users/danielapalleschi/Library/Caches/org.R-project.R/R/renv/sandbox/macos/R-4.4/aarch
```

- `[1]` is the local project library path
- `[2]` is the path to a global package cache that `renv` maintains so that you don't repeatedly download packages to your machine for each project library
  - e.g., if we already have `ggplot2` installed globally on our machine, whenever we want to add it to a project library we don't need to re-install it entirely from the CRAN (unless we want a different package version)

Installing more packages

- which packages are stored in `renv.lock`?
  - only those that are used within your project
- packages not used in your project but installed in your global library aren't included
  - to add these packages, or any other packages you want, you need to (re-)install them locally within your project

- let's install a package that we'll use later on: `lme4`

```
# as usual
install.packages("lme4")
# or with the renv package
renv::install("lme4")
```

- if you already have a package on your machine (in your global library), `renv` will just grab it from the global cache
- if not, it will be downloaded from CRAN

Installing a new package

- let's also install a package I'm confident you don't already have on your machine (as you might've already worked with `lme4` in other classes)
  - [`brms`] for Bayesian regression models using Stan

```
install.packages("brms")
renv::install("brms")
```

- and if we want a specific package version:

```
renv::install("brms@2.19.0")
```

Installing developer packages

- not all packages are available on the CRAN
  - we can install developer packages from GitHub or GitLab using, e.g., the `install_github()` function from either the `remotes` or `devtools` package (both are very common)

```
remotes::install_github("paul-buerkner/brms")
devtools::install_github("paul-buerkner/brms")
```

- *or* we can use `renv::install()`

```
# most recent version
renv::install("paul-buerkner/brms")
```

```
# a specific previous version, for which you'll need the commit ID
renv::install("paul-buerkner/brms@db6ddde90ba533cb3942bc5a62b03803773b9844")
```

Lockfile status

- you should make a habit of checking the status of your lockfile
  - you can do this by running the following:

```
renv::status()
```

- ideally, you'll usually get the following message:

```
> renv::status()
No issues found -- the project is in a consistent state.
```

- but if you've installed or updated some packages, you will get a list of any packages that are out-of-sync or haven't been stored in the lockfile (as should be our case)

Updating `renv.lock` file

- to update the lockfile and library, simply run:

```
renv::snapshot()
```

- you'll be given a list of changes to be made and asked if you want to proceed
  - if not problems are mentioned, then you can go ahead

Updating packages

- to update packages using `renv`, we can use:

```
renv::update()
# or
renv::update.packages()
```

- this will not automatically store the updated versions in the lockfile
  - to do this, include the argument `lock = TRUE`
- you can also use these functions to only check by including `check = T`

Restoring lockfile

```
renv::restore()
```

- this will restore the current project's package versions to be those stored in the lockfile
  - but only if the library was built in the same R version
  - otherwise, all packages need to be installed, and might not function the same
- useful if you
  - want to revert to the stored package versions
  - want to run your project on another computer (e.g., a collaborator)

Additional packages

- some other packages that can be useful for package management or reproducibility
- `groundhog`: version control for CRAN, GitHub, and GitLab packages
  - uses `groundhog.library()` instead of `library()` to load packages
  - can take a list of libraries (or an object which contains such a list) and a date as arguments
  - will then install the package versions that were available at the given date
- issues can arise when package versions were built on a previous version of R, and are no longer supported
  - this can cause the installation to fail (just like with `renv`)

Session Info

- whether you're using `renv` or not, *always* end a script with `sessionInfo()`
  - with dynamic reports: this will print out the package versions used to produce the output
  - in R: you can save the info as an object and save it as an RDS file
    - or run it, copy-and-paste the output in the script, and comment it all out

```
sessionInfo()
```

```
R version 4.4.0 (2024-04-24)
Platform: aarch64-apple-darwin20
Running under: macOS Ventura 13.2.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  L

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: Europe/Berlin
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices datasets  utils     methods   base

loaded via a namespace (and not attached):
 [1] digest_0.6.36     fastmap_1.2.0     xfun_0.47         magrittr_2.0.3
 [5] knitr_1.48        htmltools_0.5.8.1 rmarkdown_2.28    cli_3.6.3
 [9] renv_1.0.7        compiler_4.4.0    rprojroot_2.0.4   here_1.0.1
[13] rstudioapi_0.16.0 tools_4.4.0       evaluate_0.24.0   Rcpp_1.0.12
[17] yaml_2.3.10       magick_2.8.3      rlang_1.1.4       jsonlite_1.8.8
```

Learning objectives

Today we will…

- learn about R package repositories
- learn how package dependencies can break code
- use the `renv` package to create and maintain a project-relative package library

References

Rodrigues, B. (2023). *Building reproducible analytical pipelines with R.*