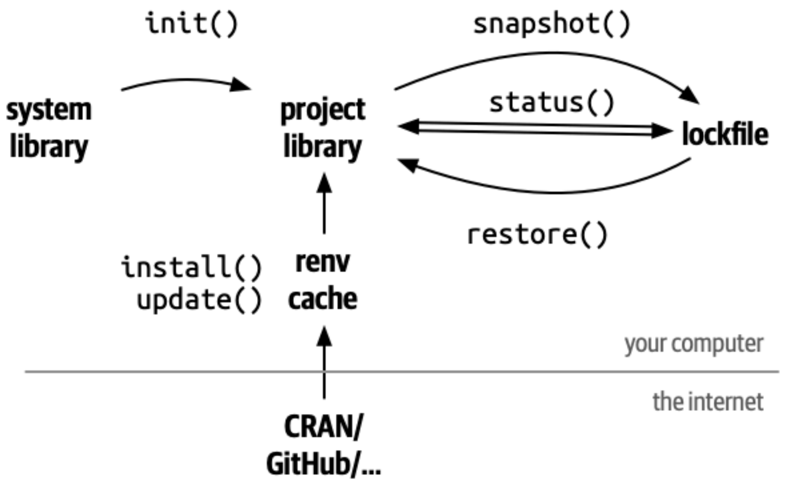
renv 1.0.7

Introduction to renv

Source: `vignettes/renv.Rmd`

The renv package helps you create reproducible environments for your R projects. This vignette introduces you to the basic nouns and verbs of renv, like the user and project libraries, and key functions like `renv::init()`, `renv::snapshot()` and `renv::restore()`. You'll also learn about some of the infrastructure that makes renv tick, some problems that renv doesn't help with, and how to uninstall it if you no longer want to use it.



The most important verbs and nouns of renv, which you'll learn more about below.

We assume you're already living a project-centric lifestyle and are familiar with a version control system, like Git and GitHub: we believe these are table stakes for reproducible data science. If you're not already using projects, we recommend Workflow: Projects from *R for Data Science*; if you're unfamiliar with Git and GitHub, we recommend Happy Git and GitHub for the useR.

understand two important pieces of R jargon: libraries and repositories.

A **library** is a directory containing installed packages. This term is confusing because you write (e.g.) `library(dplyr)`, making it easy to think that you're loading the dplyr library, not the dplyr package. That confusion doesn't usually matter because you don't have to think about libraries, simply installing all packages into a **system library**[1] that's shared across all projects. With renv, you'll start using **project libraries,** giving each project its own independent collection of packages.

You can see your current libraries with `.libPaths()` and see which packages are available in each library with `lapply(.libPaths(), list.files)`.

A **repository** is a source of packages; `install.packages()` gets a package from a repository (usually somewhere on the Internet) and puts it in a library (a directory on your computer). The most important repository is CRAN which is available to install packages from in just about every R session. Other freely available repositories include Bioconductor, the Posit Public Package Manager, and R Universe (which turns GitHub organisations into repositories).

You can see which repositories are currently set up in your session with `getOption("repos")`; when you call `install.packages("{pkgname}")`, R will look for `pkgname` in each repository in turn.

Getting started

- The project library, `renv/library`, is a library that contains all packages currently used by your project[2]. This is the key magic that makes renv work: instead of having one library containing the packages used in every project, renv gives you a separate library for each project. This gives you the benefits of **isolation**: different projects can use different versions of packages, and installing, updating, or removing packages in one project doesn't affect any other project.
- The **lockfile**, `renv.lock`, records enough metadata about every package that it can be re-installed on a new machine. We'll come back to the lockfile shortly when we talk about `renv::snapshot()` and `renv::restore()`.
- A project R profile, `.Rprofile`. This file is run automatically every time you start R (in that project), and renv uses it to configure your R session to use the project library. This ensures that once you turn on renv for a project, it stays on, until you deliberately turn it off.

The next important pair of tools are `renv::snapshot()` and `renv::restore()`. `snapshot()` updates the lockfile with metadata about the currently-used packages in the project library. This is useful because you can then share the lockfile and other people or other computers can easily reproduce your current environment by running `restore()`, which uses the metadata from the lockfile to install exactly the same version of every package. This pair of functions gives you the benefits of **reproducibility** and **portability**: you are now tracking exactly which package versions you have installed so you can recreate them on other machines.

Now that you've got the a high-level lay of the land, we'll show a couple of specific workflows before discussing some

Collaboration

One of the reasons to use renv is to make it easier to share your code in such a way that everyone gets exactly the same package versions as you. As above, you'll start by calling `renv::init()`. You'll then need to commit `renv.lock`, `.Rprofile`, `renv/settings.json` and `renv/activate.R` to version control, ensuring that others can recreate your project environment. If you're using git, this is particularly simple because renv will create a `.gitignore` for you, and you can just commit all suggested files[3].

Now when one of your collaborators opens this project, renv will automatically bootstrap itself, downloading and installing the appropriate version of renv. It will also ask them if they want to download and install all the packages it needs by running `renv::restore()`.

Installing packages

Over time, your project will need more packages. One of the philosophies of renv is that your existing package management workflows should continue to work, so you can continue to use familiar tools like `install.packages()`[4]. But you can also use `renv::install()`: it's a little less typing and can install packages from GitHub, Bioconductor, and more, not just CRAN.

If you use renv for multiple projects, you'll have multiple libraries, meaning that you'll often need to install the same package in multiple places. It would be annoying if you had to download (or worse, compile) the package repeatedly, so renv uses a package cache. That means you only ever have to download and install a package once, and for each subsequent install, renv will just add a link from the project

After installing the package and checking that your code works, you should call `renv::snapshot()` to record the latest package versions in your lockfile. If you're collaborating with others, you'll need to commit those changes to git, and let them know that you've updated the lockfile and they should call `renv::restore()` when they're next working on a project.

Updating packages

It's worth noting that there's a small risk associated with isolation: while your code will never break due to a change in another package, it will also never benefit from bug fixes. So for packages under active development, we recommend that you regularly (at least once a year) use `renv::update()` [5] to get the latest versions of all dependencies. Similarly, if you're making major changes to a project that you haven't worked on for a while, it's often a good idea to start with an `renv::update()` before making any changes to the code.

After calling `renv::update()`, you should run the code in your project and verify that it still works (or make any changes needed to get it working). Then call `renv::snapshot()` to record the new versions in the lockfile. If you get stuck, and can't get the project to work with the new versions, you can call `renv::restore()` to roll back changes to the project library and revert to the known good state recorded in your lockfile. If you need to roll back to an even older version, take a look at `renv::history()` and `renv::revert()`.

`renv::update()` will also update renv itself, ensuring that you get all the latest features. See `renv::upgrade()` if you ever want to upgrade just renv, or you need to install a development version from GitHub.

it’s time to learn a bit more about how the lockfile works. You won’t typically edit this file directly, but you’ll see it changing in your git commits, so it’s good to have a sense for what it looks like.

The lockfile is always called `renv.lock` and is a json file that records all the information needed to recreate your project in the future. Here’s an example lockfile, with the markdown package installed from CRAN and the mime package installed from GitHub:

```
{
  "R": {
    "Version": "4.3.3",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://cloud.r-project.org"
      }
    ]
  },
  "Packages": {
    "markdown": {
      "Package": "markdown",
      "Version": "1.0",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "4584a57f565dd7987d59dda3a02cfb41"
    },
    "mime": {
      "Package": "mime",
      "Version": "0.12.1",
      "Source": "GitHub",
      "RemoteType": "github",
      "RemoteHost": "api.github.com",
      "RemoteUsername": "yihui",
      "RemoteRepo": "mime",
      "RemoteRef": "main",
      "RemoteSha": "1763e0dcb72fb58d97bab97bb834fc71
```

```
          "Hash": "c2772b6269924dad6784aaa1d99dbb86"
        }
      }
    }
```

As you can see the json file has two main components: `R` and `Packages`. The `R` component contains the version of R used, and a list of repositories where packages were installed from. The `Packages` contains one record for each package used by the project, including all the details needed to re-install that exact version. The fields written into each package record are derived from the installed package's `DESCRIPTION` file, and include the data required to recreate installation, regardless of whether the package was installed from CRAN, Bioconductor, GitHub, Gitlab, Bitbucket, or elsewhere. You can learn more about the sources renv supports in `vignette("package-sources")`.

Caveats

It is important to emphasize that renv is not a panacea for reproducibility. Rather, it is a tool that can help make projects reproducible by helping with one part of the overall problem: R packages. There are a number of other pieces that renv doesn't currently provide much help with:

- **R version**: renv tracks, but doesn't help with, the version of R used with the package. renv can't easily help with this because it's run inside of R, but you might find tools like rig helpful, as they make it easier to switch between multiple version of R on one computer.
- **Pandoc**: The rmarkdown package relies heavily on pandoc, but pandoc is not bundled with the rmarkdown package. That means restoring rmarkdown from the lockfile is insufficient to guarantee exactly the same

- **Operating system, versions of system libraries, compiler versions**: Keeping a 'stable' machine image is a separate challenge, but Docker is one popular solution. See `vignette("docker", package = "renv")` for recommendations on how Docker can be used together with renv.

You also need to be aware that package installation may fail if a package was originally installed through a binary, but that binary is no longer available. renv will attempt to install the package from source, but this can (and often will) fail due to missing system prerequisites.

Ultimately, making a project reproducible will always require thought, not just mechanical usage of a tool: what does it mean for a particular project to be reproducible, and how can you use tools to meet that particular goal of reproducibility?

Uninstalling renv

If you find renv isn't the right fit for your project, deactivating and uninstalling it is easy.

- To deactivate renv in a project, use `renv::deactivate()`. This removes the renv auto-loader from the project `.Rprofile`, but doesn't touch any other renv files used in the project. If you'd like to later re-activate renv, you can do so with `renv::activate()`.

- To completely remove renv from a project, call `renv::deactivate(clean = TRUE)`. If you later want to use renv for this project, you'll need to start from scratch with `renv::init()`.

```
root <- renv::paths$root()
unlink(root, recursive = TRUE)
```

You can then uninstall the renv package with `utils::remove.packages("renv")`.

ON THIS PAGE