Managing Packages and Environments

Childhood Cancer Data Lab

Software is called “soft” for a reason

- Software is always changing!
  - New versions can bring new features and fix bugs 🎉
  - But also remove features you relied on 😟
  - Or alter behavior in unexpected ways 😱

- Changes in one piece of software can break other software
  - Sometimes this is intentional! Operating systems are constantly updating to break hacking tools

Changes occur at every level of the computing “stack”

- Individual scripts/analyses
- Packages within R, Python, etc.
- Individual programs (Cell Ranger, Salmon, etc., but also R & Python)
- Operating system
- Hardware

We want to do our best to **track** and **document** versions of as many layers as possible.

Ideally, we would like to **freeze** versions, so we and anyone else can come back and know results will be the same!

The “analysis” layer

- We have already talked about tracking your changes with Git and GitHub
  - If you know which commit of your scripts you used to produce an analysis, you can point people right to that
  - “**tags**” and “**releases**” on GitHub can make this easier when you have a particular commit you want to share (but we won’t be covering that in this workshop)

The package layer

- Research software tools in bioinformatics (and beyond) are often published as packages for R or Python
  - Examples: Seurat, scanpy, tidyverse, pandas, Bioconductor packages
- This makes them generally easy to install and update as research progresses
  - BUT easy to update means things can change fast
  - New versions may change results, even for existing functions!
    - Newer isn't always better; sometime you want to stick with the old way
  - Dependencies on other packages may require specific versions of other packages

Documenting package versions in R

- `sessionInfo()` is your friend
  - or `sessioninfo::session_info()`

```
R version 4.1.2 (2021-11-01)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Monterey 12.4

Matrix products: default
LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Reso

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

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

other attached packages:
[1] magrittr_2.0.3 ggplot2_3.3.6  dplyr_1.0.9

loaded via a namespace (and not attached):
 [1] bslib_0.3.1        jquerylib_0.1.4    RColorBrewer_1
 [5] compiler_4.1.2     tools_4.1.2        digest_0.6.29
 [9] jsonlite_1.8.0     evaluate_0.15      lifecycle_1.0.
[13] gtable_0.3.0       pkgconfig_2.0.3    rlang_1.0.2
[17] rstudioapi_0.13    yaml_2.3.5         parallel_4.1.2
```

```
─ Session info ──────────────────────────────────────────────────────────────
 setting  value
 version  R version 4.1.2 (2021-11-01)
 os       macOS Monterey 12.4
 system   x86_64, darwin17.0
 ui       RStudio
 language (EN)
 collate  en_US.UTF-8
 ctype    en_US.UTF-8
 tz       America/New_York
 date     2022-05-31
 rstudio  2022.02.2+485 Prairie Trillium (desktop)
 pandoc   2.17.1.1 @ /Applications/RStudio.app/Contents/MacOS/quarto/bin/ (via rmarkdown)

─ Packages ──────────────────────────────────────────────────────────────────
 package      * version date (UTC) lib source
 bit            4.0.4   2020-08-04 [1] CRAN (R 4.1.0)
 bit64          4.0.5   2020-08-30 [1] CRAN (R 4.1.0)
 bslib          0.3.1   2021-10-06 [1] CRAN (R 4.1.0)
 cli            3.3.0   2022-04-25 [1] CRAN (R 4.1.2)
 colorspace     2.0-3   2022-02-21 [1] CRAN (R 4.1.2)
 crayon         1.5.1   2022-03-26 [1] CRAN (R 4.1.2)
 digest         0.6.29  2021-12-01 [1] CRAN (R 4.1.0)
 dplyr        * 1.0.9   2022-04-28 [1] CRAN (R 4.1.2)
 ellipsis       0.3.2   2021-04-29 [1] CRAN (R 4.1.0)
 evaluate       0.15    2022-02-18 [1] CRAN (R 4.1.2)
```

But how do you recreate the same set of packages?

Installing packages based on `sessionInfo()` output could be very tedious!

Enter **`renv`**!

- **`renv`** is an R package for *tracking*, *freezing*, and *sharing* R environments, including all of the package versions that were installed.
- Each project can have its own environment, with its own set of packages
  - Different projects may require different versions of packages
  - **`renv`** can help manage these different sets of package versions
- When sharing a project/analysis, using **`renv`** allows everyone stay in sync with same packages and versions

How does `renv` work?

Rather than using the system R package library, renv creates a library for each project that R will use when running code for the project ("Project Library")

This `renv`-created library could be large, so we can't reasonably share the whole thing.

Instead, we create a file (`renv.lock`) that describes the library.

`renv` uses this file to track all of the packages we are using, and recreate the library with those packages as needed.

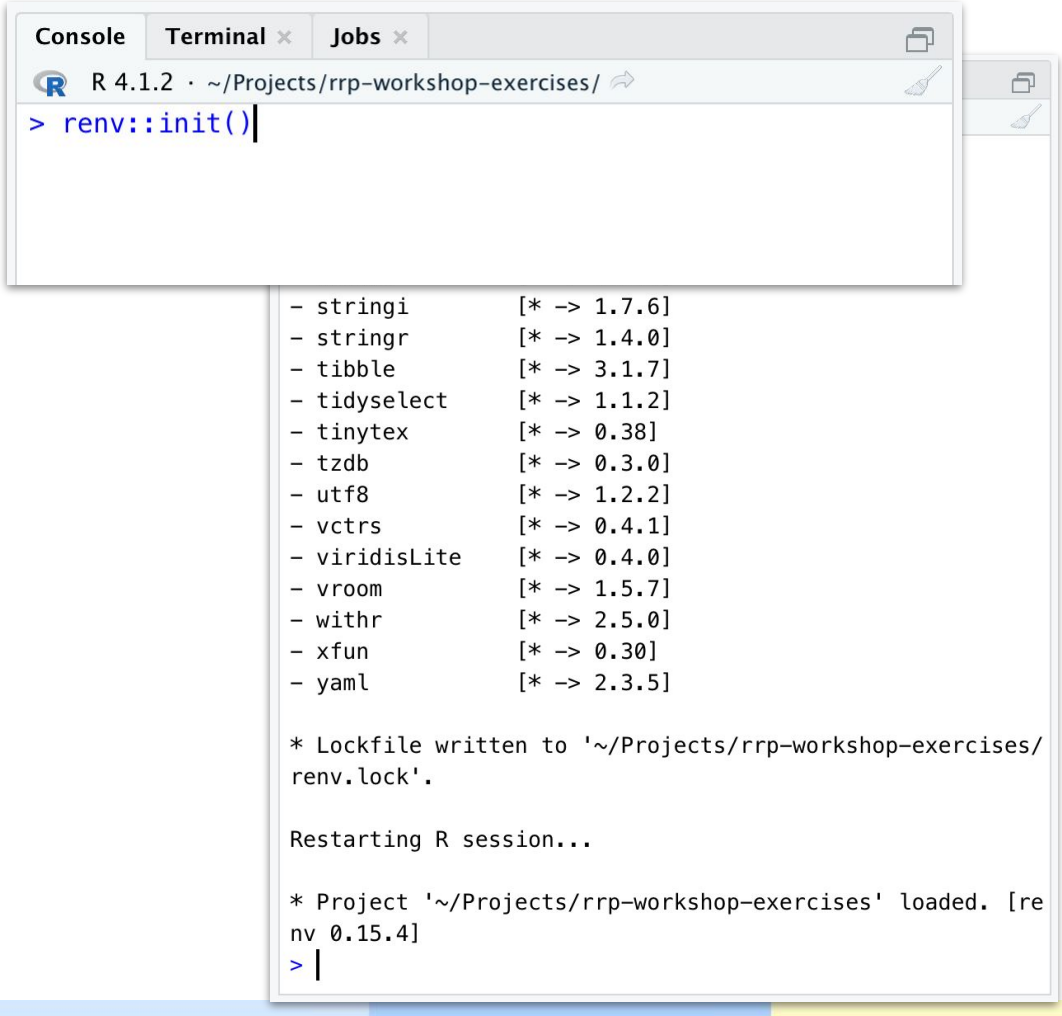
renv initialization

In the console, enter:

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

Lots of text will scroll by, and your R session will restart.

That's it! You are starting to track your R packages!



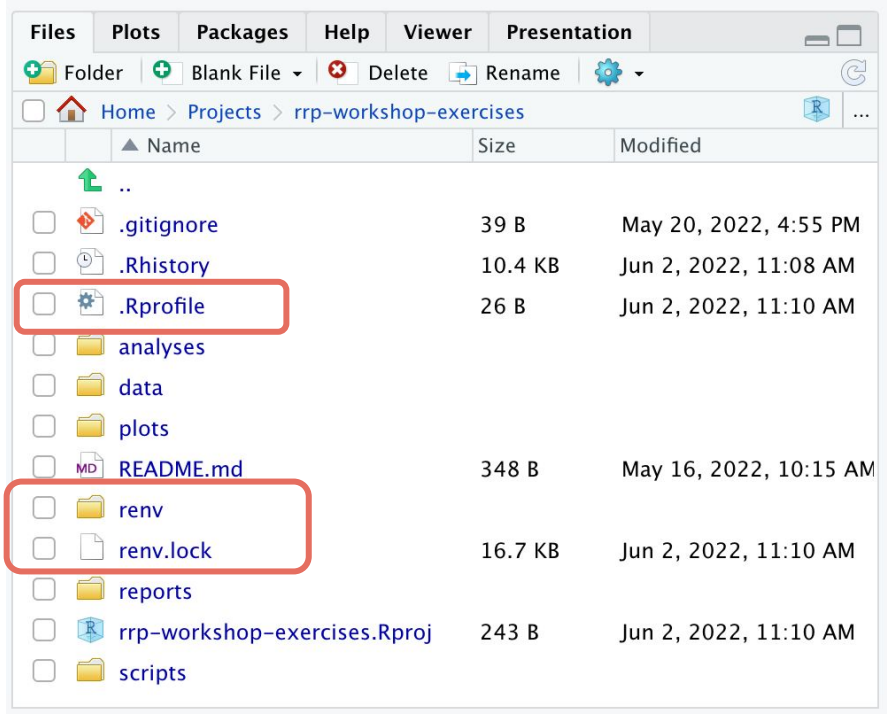
What did `renv::init()` do?

Added an **`renv.lock`** file, **`renv/`** folder and **`.Rprofile`** file
(or modified the one you had)

The **`.Rprofile`** file is run when R launches *for this project*, and it contains a command to configure `renv` on launch.

The **`renv/`** folder is where the Project Library and support files can be found

Newly installed packages for the project will be stored in this Project Library



The `renv.lock` file

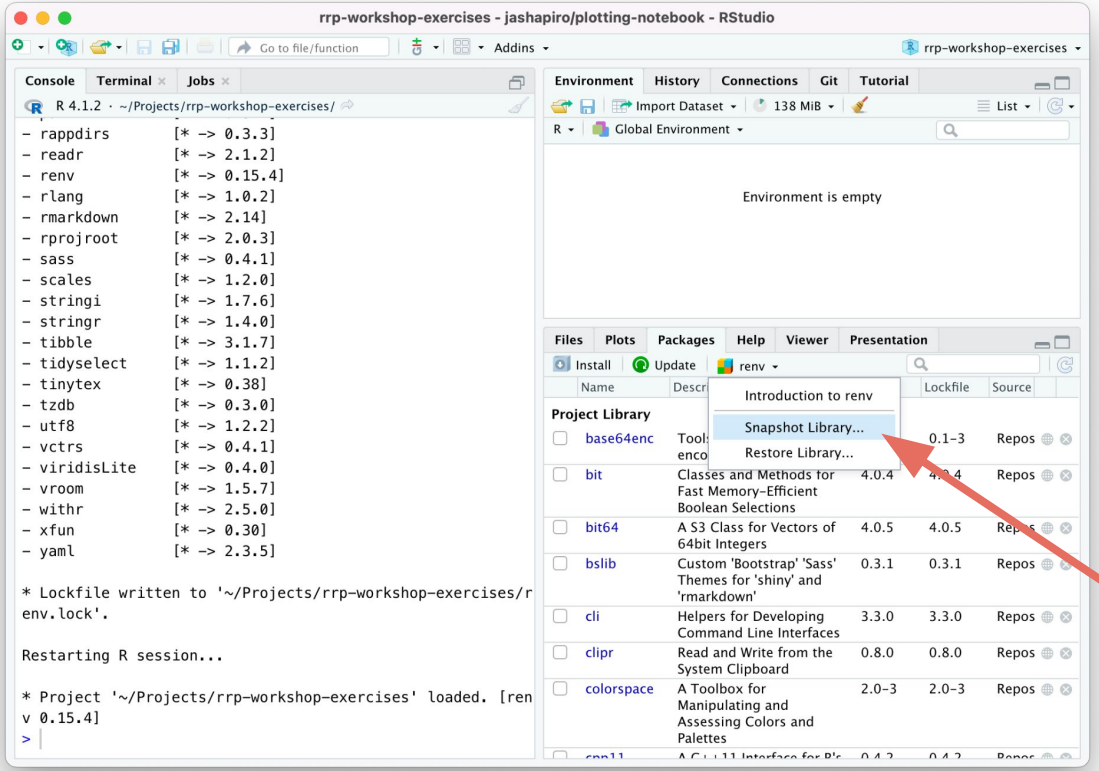
Taking a snapshot creates or updates the `renv.lock` file at the base of your project.

This file records…

- Which packages are installed
- The package versions
- Where the packages came from

Do not edit this file manually!

Updating the `renv.lock` file



An "renv" menu now appears in the Packages pane

Use "Snapshot Library..." to update the `renv.lock` file to the current setup, e.g. after you update or install new packages

Alternatively, in the console enter:

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

Restoring a library from an `renv.lock` file

When working on a new machine, or if someone else updated the `renv.lock` file, you may need to update the Project Library

```
* Project '~/Projects/rrp-workshop-exercises' loaded. [renv 0.15.4]
* The project library is out of sync with the lockfile.
* Use `renv::restore()` to install packages recorded in the lockfile.
```

Follow the instructions! Enter **`renv::restore()`** in the console (or use the renv menu "Restore Library" option) to sync your Package Library with the recorded versions, installing any missing packages.

Which packages are included in `renv.lock`?

You might have many packages installed, but only use some in a given project.

`renv` tries to be smart about this, and only includes packages that it finds ***used*** within code in the project folder, or packages that are required by the packages that are used (so-called *dependencies*)

Sometimes renv misses a package (particularly for packages with optional dependencies), and you might need to create a file (we usually call ours `dependencies.R`) that only contains lines like:

```
library(missing_package)
```

which will force renv to include that package in the lockfile.

We need more than renv sometimes: enter **conda**

- As long as we are using R, **renv** is usually sufficient for package management
  - But lots of tools are written in Python, and other programming languages
  - Many languages have their own package managers: **pip** for Python, **rbenv** for Ruby, etc.
  - We also care about versions of binary packages, like **salmon** or **bwa**
  - It can be a lot to keep track of!
- **conda** can handle it all (usually)
  - Started as a Python package manager, but it can be used for just about any command line software
  - Like renv, you can create separate sets of software with different versions for different projects
  - LOTS of bioinformatics software available through **bioconda**
  - https://bioconda.github.io/user/install.html

Advantages of Conda

- Userspace installation!
  - Everything is installed in *your* user account, so you don't need administrative privileges on the machine
  - This is great for shared servers, like many HPC grids
- Multiple environments
  - You can have different versions of software packages in different environments
    - Leave the OS Python alone but use a newer version which has new features or better performance (Python 3.11 is a bug speed bump!)
    - Great for testing & legacy support
  - Software with conflicting dependencies can live in different environments
- Well-supported ecosystem, with a huge number of supported packages

Conda terminology

- Environment
  - A set of software and packages, with specific versions for everything that is installed
  - You can have many!
  - Best practice is going to be one environment for each project that you work on
- Package
  - An individual software tool or library
  - Often has *dependencies*: other packages that are needed for this package to run
- Channel
  - An upstream repository of packages that you can download software from
  - **conda-forge** and **Bioconda** are the two we use most often
    (the official Anaconda **defaults** channel used to be on this list, but they may ask for license fees if you use it)

Creating an environment

To create an environment, minimally:

```
$ conda create --name myenv
```

But this is not usually all you want… we can also specify packages to include, optionally with versions:

```
$ conda create --name myenv python=3.11 pandas
```

Activating an environment

```
$ conda activate myenv
(myenv) $
```

- Once you have activated an environment, you can add more packages

```
(myenv) $ conda install bwa
```

- When you are done, you can deactivate the environment

```
(myenv) $ conda deactivate
$
```

* `deactivate` actually will take you back to the previous active environment, often `base`

What if you have a lot of packages to include?

Use an `environment.yml` file with the packages listed in it:

```
$ conda env create --name myenv --file environment.yml
```

Why do you need `env` here? Good question.

What is in an `environment.yml` file?

- `name`: the environment name
- `channels`: where the packages come from
- `dependencies`: what packages are needed
    - And optionally, what version of those packages

```yaml
name: openscpca
channels:
  - conda-forge
  - bioconda
  - defaults
dependencies:
  - python=3.11
  - awscli>=2.15
  - conda-lock>=2.5
  - jq>=1.7
  - pre-commit>=3.6
```

Saving an environment for sharing

- One way: From the environment you are in:

```
$ conda export > environment.yml
```

- But this includes everything currently installed with build hashes, and may be more than you need
- Often more useful:

```
$ conda export --from-history --no-builds > environment.yml
```

Conda environment tips

- Keep your base environment lean
  - You might install things you use all the time (like your favorite version of Python)
  - But: having as little as possible in the environment avoids conflicts
- One environment per project is a good starting point
  - You might have more if you need to avoid conflicts, or just want to keep things more modular
- Using Jupyter? Install it in your environment and launch from that activated environment
  - It *is* possible to change conda environments from within Jupyter, but launching from the correct environment is usually easier & ensures all requirements are installed
- Don't worry about installing the same thing in multiple environments
  - Conda is pretty smart about using links so you won't have many copies of the same files

Can I interest you in containers?

- ***renv*** is pretty useful, but it only really helps if we are using R
  (and are careful about updating our R version)
- ***conda*** is good too, but sometimes still runs into cross-platform incompatibilities
- ***Docker*** and ***Singularity*** are another level up
  - **Containers** include *everything* from the operating system up
  - Run one OS inside another, with *all the things* frozen to particular versions
  - Once a container is built, you can distribute that directly (not just the specification file)
  - Cloud platforms love containers…

Locking an environment

- Conda's dirty secret: different systems (Mac, Linux, Windows) have different packages and dependencies.
  - An environment file from one may not work on another
  - The more detail that is specified, the less likely it is to work on a different system
  - This… sucks
- Enter **conda-lock**!
  - **conda-lock** lets us keep our `environment.yml` file high-level
  - Specify the tools and optionally which versions we need, but not everything that those requirements may depend on
  - Let conda-lock record exactly what was used, and what would be needed on each other system

```
$ conda-lock --file environment.yml
```

  - Records results in `conda-lock.yml`