# Package 'renv'

July 21, 2021

**Type** Package

**Title** Project Environments

**Version** 0.14.0

**Description** A dependency management toolkit for R. Using 'renv', you can create and manage project-local R libraries, save the state of these libraries to a 'lockfile', and later restore your library as required. Together, these tools can help make your projects more isolated, portable, and reproducible.

**License** MIT + file LICENSE

**URL** https://rstudio.github.io/renv/

**BugReports** https://github.com/rstudio/renv/issues

**Imports** utils

**Suggests** R6, BiocManager, cli, covr, devtools, knitr, miniUI, packrat, remotes, reticulate, rmarkdown, rstudioapi, shiny, testthat, uuid, yaml

**Encoding** UTF-8

**RoxygenNote** 7.1.1

**VignetteBuilder** knitr

**NeedsCompilation** no

**Author** Kevin Ushey [aut, cre], RStudio, PBC [cph]

**Maintainer** Kevin Ushey <kevin@rstudio.com>

**Repository** CRAN

**Date/Publication** 2021-07-21 21:00:02 UTC

## R topics documented:

## renv-package *Project-local Environments for R*

### Description

Project-local environments for R.

### Details

You can use `renv` to construct isolated, project-local R libraries. Each project using `renv` will share package installations from a global cache of packages, helping to avoid wasting disk space on multiple installations of a package that might otherwise be shared across projects.

### Author(s)

**Maintainer**: Kevin Ushey <kevin@rstudio.com>

Other contributors:

- RStudio, PBC [copyright holder]

### See Also

Useful links:

- https://rstudio.github.io/renv/
- Report bugs at https://github.com/rstudio/renv/issues

## activate *Activate a Project*

### Description

Activate a project, thereby loading it in the current session and also writing the infrastructure necessary to ensure the project is auto-loaded for newly-launched R sessions.

### Usage

```
activate(project = NULL, profile = NULL)
```

### Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| `profile` | The profile to be activated. When `NULL`, the default profile is activated instead. See `vignette("profiles",package = "renv")` for more information. |

## Details

Using `activate()` will:

1. Load the requested project via `load()`,
2. Add `source("renv/init.R")` to the project `.Rprofile`, thereby instructing newly-launched R sessions to automatically load the current project.

Normally, `activate()` is called as part of `init()` when a project is first initialized. However, `activate()` can be used to activate (or re-activate) an `renv` project – for example, if the project was shared without the auto-loader included in the project `.Rprofile`, or because that project was previously deactivated (via `deactivate()`).

## Value

The project directory, invisibly. Note that this function is normally called for its side effects.

## See Also

Other renv: `deactivate()`

## Examples

```
## Not run:

# activate the current project
renv::activate()

# activate a separate project
renv::activate("~/projects/analysis")


## End(Not run)
```

---

# checkout *Checkout a Repository*

---

## Description

`renv::checkout()` can be used to install and use the latest packages available from the requested repositories. This can be useful for cleaning up a library which has become a mish-mash of packages installed from a variety of disparate sources.

### Usage

```
checkout(
  repos = getOption("repos"),
  ...,
  packages = NULL,
  clean = FALSE,
  project = NULL
)
```

### Arguments

| | |
|---|---|
| repos | The R package repositories to check out. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |
| packages | The packages to be installed. When NULL (the default), all packages currently used in the project will be installed. |
| clean | Boolean; remove packages not recorded in the lockfile from the target library? Use clean = TRUE if you'd like the library state to exactly reflect the lockfile contents after restore(). |
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

---

## clean *Clean a Project*

---

### Description

Clean up a project and its associated R libraries.

### Usage

```
clean(project = NULL, ..., actions = NULL, prompt = interactive())
```

### Arguments

| | |
|---|---|
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |
| actions | The set of clean actions to take. See the documentation in **Actions** for a list of available actions, and the default actions taken when no actions are supplied. |
| prompt | Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt. |

## Value

The project directory, invisibly. Note that this function is normally called for its side effects.

## Actions

The following clean actions are available:

`package.locks` During package installation, R will create package locks in the library path, typically named 00LOCK-<package>. On occasion, if package installation fails or R is terminated while installing a package, these locks can be left behind and will inhibit future attempts to reinstall that package. Use this action to remove such left-over package locks.

`library.tempdirs` During package installation, R may create temporary directories with names of the form file\w{12}, and on occasion those files can be left behind even after they are no longer in use. Use this action to remove such left-over directories.

`system.library` In general, it is recommended that only packages distributed with R are installed into the system library (the library path referred to by `.Library`). Use this action to remove any user-installed packages that have been installed to the system library.

Because this action is destructive, it is by default never run – it must be explicitly requested by the user.

`unused.packages` Remove packages that are installed in the project library, but no longer appear to be used in the project sources.

Because this action is destructive, it is by default only run in interactive sessions when prompting is enabled.

## Examples

```
## Not run:

# clean the current project
renv::clean()


## End(Not run)
```

---

# config *User-Level Configuration of renv*

---

## Description

Configure different behaviors of `renv`.

## Usage

```
config
```

## Format

An object of class `list` of length 32.

## Details

For a given configuration option:

1. If an R option of the form renv.config.<name> is available, then that option's value will be used;
2. If an environment variable of the form RENV_CONFIG_<NAME> is available, then that option's value will be used;
3. Otherwise, the default for that particular configuration value is used.

Any periods (`.`)s in the option name are transformed into underscores (_) in the environment variable name, and vice versa. For example, the configuration option `auto.snapshot` could be configured as:

- options(renv.config.auto.snapshot = <...>)
- Sys.setenv(RENV_CONFIG_AUTO_SNAPSHOT = <...>)

Note that if both the R option and the environment variable are defined, the R option will be used instead. Environment variables can be more useful when you want a particular configuration to be automatically inherited by child processes; if that behavior is not desired, then the R option may be preferred.

If you want to set and persist these options across multiple projects, it is recommended that you set them in your user startup files (e.g. in ~/.Rprofile or ~/.Renviron).

## Configuration

The following `renv` configuration options are available:

`renv.config.activate.prompt` Automatically prompt the user to activate the current project, if it does not appear to already be activated? This is mainly useful to help ensure that calls to `renv::snapshot()` and `renv::restore()` use the project library. See `?renv::activate` for more details. Defaults to `TRUE`.

`renv.config.auto.snapshot` Automatically snapshot changes to the project library after a new package is installed? Defaults to `FALSE`.

`renv.config.bitbucket.host` The default Bitbucket host to be used during package retrieval. Defaults to `"api.bitbucket.org/2.0"`.

`renv.config.copy.method` The method to use when attempting to copy directories. See **Copy Methods** for more information. Defaults to `"auto"`.

`renv.config.connect.timeout` The amount of time to spend (in seconds) when attempting to download a file. Only applicable when the `curl` downloader is used. Defaults to `20L`.

`renv.config.connect.retry` The number of times to attempt re-downloading a file, when transient download errors occur. Only applicable when the `curl` downloader is used. Defaults to `3L`.

`renv.config.cache.enabled` Enable the global `renv` package cache? When active, `renv` will install packages into a global cache, and link or copy packages from the cache into your R library as appropriate. This can greatly save on disk space and install time when R packages are shared across multiple projects in the same environment. Defaults to `TRUE`.

`renv.config.cache.symlinks` Symlink packages from the global `renv` package cache into your project library? When `TRUE`, `renv` will use symlinks (or, on Windows, junction points) to reference packages installed in the cache. Set this to `FALSE` if you'd prefer to copy packages from the cache into your project library. Defaults to `TRUE`.

`renv.config.dependency.errors` Many `renv` APIs require the enumeration of your project's R package dependencies. This option controls how errors that occur during this enumeration are handled. By default, errors are reported but are non-fatal. Set this to `"fatal"` to force errors to be fatal, and `"ignored"` to ignore errors altogether. See `dependencies()` for more details. Defaults to `"reported"`.

`renv.config.exported.functions` When `library(renv)` is called, should its exports be placed on the search path? Set this to `FALSE` to avoid issues that can arise with, for example, `renv::load()` masking `base::load()`. In general, we recommend referencing `renv` functions from its namespace explicitly; e.g. prefer `renv::snapshot()` over `snapshot()`. By default, all exported `renv` functions are attached and placed on the search path, for backwards compatibility with existing scripts using `renv`. Defaults to `"*"`.

`renv.config.external.libraries` A character vector of external libraries, to be used in tandem with your projects. Be careful when using external libraries: it's possible that things can break within a project if the version(s) of packages used in your project library happen to be incompatible with packages in your external libraries; for example, if your project required xyz 1.0 but xyz 1.1 was present and loaded from an external library. Can also be an R function that provides the paths to external libraries. Library paths will be expanded via `.expand_R_libs_env_var()` as necessary. Defaults to `NULL`.

`renv.config.filebacked.cache` Enable the `renv` file-backed cache? When enabled, `renv` will cache the contents of files that are read (e.g. DESCRIPTION files) in memory, thereby avoiding re-reading the file contents from the filesystem if the file has not changed. `renv` uses the file `mtime` to determine if the file has changed; consider disabling this if `mtime` is unreliable on your system. Defaults to `TRUE`.

`renv.config.github.host` The default GitHub host to be used during package retrieval. Defaults to `"api.github.com"`.

`renv.config.gitlab.host` The default GitLab host to be used during package retrieval. Defaults to `"gitlab.com"`.

`renv.config.hydrate.libpaths` A character vector of library paths, to be used by `hydrate()` when attempting to hydrate projects. When empty, the default set of library paths (as documented in `?hydrate`) are used instead. See `hydrate()` for more details. Defaults to `NULL`.

`renv.config.install.shortcuts` Allow for a set of 'shortcuts' when installing packages with renv? When enabled, if `renv` discovers that a package to be installed is already available within the user or site libraries, then it will install a local copy of that package. Defaults to `TRUE`.

`renv.config.install.staged` DEPRECATED: Please use `renv.config.install.transactional` instead. Defaults to `TRUE`.

`renv.config.install.transactional` Perform a transactional install of packages during install and restore? When enabled, `renv` will first install packages into a temporary library, and

later copy or move those packages back into the project library only if all packages were successfully downloaded and installed. This can be useful if you'd like to avoid mutating your project library if installation of one or more packages fails. Defaults to TRUE.

`renv.config.install.verbose` Be verbose when installing R packages from sources? When TRUE, renv will stream any output generated during package build + installation to the console. Defaults to FALSE.

`renv.config.locking.enabled` Use interprocess locks when invoking methods which might mutate the project library? Enable this to allow multiple processes to use the same renv project, while minimizing risks relating to concurrent access to the project library. Disable this if you encounter locking issues. Locks are stored as files within the project at renv/lock; if you need to manually remove a stale lock you can do so via `unlink("renv/lock",recursive = TRUE)`. Defaults to FALSE.

`renv.config.mran.enabled` Attempt to download binaries from MRAN during restore? See `vignette("mran",package = "renv")` for more details. Defaults to TRUE.

`renv.config.repos.override` Override the R package repositories used during `restore()`? Primarily useful for deployment / continuous integration, where you might want to enforce the usage of some set of repositories over what is defined in renv.lock or otherwise set by the R session. Defaults to NULL.

`renv.config.rspm.enabled` Boolean; enable RSPM integration for renv projects? When TRUE, renv will attempt to transform the repository URLs used by RSPM into binary URLs as appropriate for the current platform. Set this to FALSE if you'd like to continue using source-only RSPM URLs, or if you find that renv is improperly transforming your repository URLs. Defaults to TRUE.

`renv.config.sandbox.enabled` Enable sandboxing for renv projects? When active, renv will attempt to sandbox the system library, preventing non-system packages installed in the system library from becoming available in renv projects. (That is, only packages with priority "base" or "recommended", as reported by `installed.packages()`, are made available.) Sandboxing is done by linking or copying system packages into a separate library path, and then instructing R to use that library path as the system library path. In some environments, this action can take a large amount of time – in such a case, you may want to disable the renv sandbox. Defaults to TRUE.

`renv.config.shims.enabled` Should renv shims be installed on package load? When enabled, renv will install its own shims over the functions `install.packages()`, `update.packages()` and `remove.packages()`, delegating these functions to `install()`, `update()` and `remove()` as appropriate. Defaults to TRUE.

`renv.config.snapshot.validate` Validate R package dependencies when calling snapshot? When TRUE, renv will attempt to diagnose potential issues in the project library before creating renv.lock – for example, if a package installed in the project library depends on a package which is not currently installed. Defaults to TRUE.

`renv.config.startup.quiet` Be quiet during startup? When set, renv will not display the typical Project <path> loaded. [renv <version>] banner on startup. Defaults to NULL.

`renv.config.synchronized.check` Check that the project library is synchronized with the lockfile on load? Defaults to TRUE.

`renv.config.updates.check` Check for package updates when the session is initialized? This can be useful if you'd like to ensure that your project lockfile remains up-to-date with packages as they are released on CRAN. Defaults to FALSE.

`renv.config.updates.parallel` Check for package updates in parallel? This can be useful when a large number of packages installed from non-CRAN remotes are installed, as these packages can then be checked for updates in parallel. Defaults to `2L`.

`renv.config.user.library` Include the user library on the library paths for your projects? Note that this risks breaking project encapsulation and is not recommended for projects which you intend to share or collaborate on with other users. See also the caveats for the `renv.config.external.libraries` option. Defaults to `FALSE`.

`renv.config.user.profile` Load the user R profile (typically located at ~/.Rprofile) when `renv` is loaded? Consider disabling this if you require extra encapsulation in your projects; e.g. if your `.Rprofile` attempts to load packages that you might not install in your projects. Defaults to `FALSE`.

## Copy Methods

If you find that `renv` is unable to copy some directories in your environment, you may want to try setting the `copy.method` option. By default, `renv` will try to choose a system tool that is likely to succeed in copying files on your system – `robocopy` on Windows, and `cp` on Unix. `renv` will also instruct these tools to preserve timestamps and attributes when copying files. However, you can select a different method as appropriate.

The following methods are supported:

| | |
|---|---|
| `auto` | Use `robocopy` on Windows, and `cp` on Unix-alikes. |
| `R` | Use R's built-in `file.copy()` function. |
| `cp` | Use `cp` to copy files. |
| `robocopy` | Use `robocopy` to copy files. (Only available on Windows.) |
| `rsync` | Use `rsync` to copy files. |

You can also provide a custom copy method if required; e.g.

```
options(renv.config.copy.method = function(src, dst) {
  # copy a file from 'src' to 'dst'
})
```

Note that `renv` will always first attempt to copy a directory first to a temporary path within the target folder, and then rename that temporary path to the final target destination. This helps avoid issues where a failed attempt to copy a directory could leave a half-copied directory behind in the final location.

## Project-Local Settings

For settings that should persist alongside a particular project, the various settings available in settings can be used.

## Examples

```
# disable automatic snapshots
```

```
options(renv.config.auto.snapshot = FALSE)

# disable with environment variable
Sys.setenv(RENV_CONFIG_AUTO_SNAPSHOT = FALSE)
```

---

`consent` *Consent to usage of renv*

---

## Description

Provide consent to `renv`, allowing it to write and update certain files on your filesystem.

## Usage

```
consent(provided = FALSE)
```

## Arguments

| | |
|---|---|
| `provided` | The default provided response. If you need to provide consent from a non-interactive R session, you can invoke `renv::consent(provided = TRUE)` explicitly. |

## Details

As part of its normal operation, `renv` will write and update some files in your project directory, as well as an application-specific cache directory. These paths are documented within paths.

In accordance with the CRAN Repository Policy, `renv` must first obtain consent from you, the user, before these actions can be taken. Please call `renv::consent()` first to provide this consent.

You can also set the R option:

```
options(renv.consent = TRUE)
```

to implicitly provide consent for e.g. non-interactive R sessions.

## Value

`TRUE` if consent is provided, or an R error otherwise.

## deactivate *Deactivate a Project*

### Description

Use `deactivate()` to remove the infrastructure used by `renv` to activate projects for newly-launched R sessions. In particular, this implies removing the requisite code from the project `.Rprofile` that automatically activates the project when new R sessions are launched in the project directory.

### Usage

```
deactivate(project = NULL)
```

### Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

### Value

The project directory, invisibly. Note that this function is normally called for its side effects.

### See Also

Other renv: `activate()`

### Examples

```
## Not run:

# deactivate the currently-activated project
renv::deactivate()


## End(Not run)
```

## dependencies *Find R Package Dependencies in a Project*

### Description

Find R packages used within a project.

## Usage

```
dependencies(
  path = getwd(),
  root = NULL,
  ...,
  progress = TRUE,
  errors = c("reported", "fatal", "ignored"),
  dev = FALSE
)
```

## Arguments

| | |
|---|---|
| path | The path to a (possibly multi-mode) R file, or a directory containing such files. By default, all files within the current working directory are checked, recursively. |
| root | The root directory to be used for dependency discovery. Defaults to the active project directory. You may need to set this explicitly to ensure that your project's .renvignores (if any) are properly handled. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |
| progress | Boolean; report progress output while enumerating dependencies? |
| errors | How should errors that occur during dependency enumeration be handled? See **Errors** for more details. |
| dev | Boolean; include 'development' dependencies as well? That is, packages which may be required during development but are unlikely to be required during runtime for your project. By default, only runtime dependencies are returned. |

## Details

dependencies() will crawl files within your project, looking for R files and the packages used within those R files. This is done primarily by parsing the code and looking for calls of the form:

- library(package)
- require(package)
- requireNamespace("package")
- package::method()

For R package projects, dependencies expressed in the DESCRIPTION file will also be discovered. Note that the rmarkdown package is required in order to crawl dependencies in R Markdown files.

## Value

An R data.frame of discovered dependencies, mapping inferred package names to the files in which they were discovered.

### Suppressing Errors

Depending on how you've structured your code, `renv` may emit errors when attempting to enumerate dependencies within `.Rmd` / `.Rnw` documents. For code chunks that you'd explicitly like `renv` to ignore, you can include `renv.ignore=TRUE` in the chunk header. For example:

```
```{r chunk-label, renv.ignore=TRUE}
# code in this chunk will be ignored by renv
```
```

Similarly, if you'd like `renv` to parse a chunk that is otherwise ignored (e.g. because it has `eval=FALSE` as a chunk header), you can set:

```
```{r chunk-label, eval=FALSE, renv.ignore=FALSE}
# code in this chunk will _not be ignored
```
```

### Ignoring Files

By default, `renv` will read your project's `.gitignore`s (if any) to determine whether certain files or folders should be included when traversing directories. If preferred, you can also create a `.renvignore` file (with entries of the same format as a standard `.gitignore` file) to tell `renv` which files to ignore within a directory. If both `.renvignore` and `.gitignore` exist within a folder, the `.renvignore` will be used in lieu of the `.gitignore`.

See https://git-scm.com/docs/gitignore for documentation on the `.gitignore` format. Some simple examples here:

```
# ignore all R Markdown files
*.Rmd

# ignore all data folders
data/

# ignore only data folders from the root of the project
/data/
```

### Errors

`renv`'s attempts to enumerate package dependencies in your project can fail – most commonly, because of parse errors in your R code. The `errors` parameter can be used to control how `renv` responds to errors that occur.

| Name | Action |
|---|---|
| `"reported"` | Errors are reported to the user, but are otherwise ignored. |
| `"fatal"` | Errors are fatal and stop execution. |
| `"ignored"` | Errors are ignored and not reported to the user. |

Depending on the structure of your project, you may want `renv` to ignore errors that occur when

attempting to enumerate dependencies. However, a more robust solution would be to use an `.renvignore` file to tell `renv` not to scan such files for dependencies, or to configure the project to require explicit dependency management (`renv::settings$snapshot.type("explicit")`) and enumerate your dependencies in a project `DESCRIPTION` file.

### Development Dependencies

`renv` attempts to distinguish between 'development' dependencies and 'runtime' dependencies. For example, you might rely on e.g. devtools and roxygen2 during development for a project, but may not actually require these packages at runtime.

### Examples

```
## Not run:

# find R package dependencies in the current directory
renv::dependencies()


## End(Not run)
```

---

## `diagnostics` *Print a Diagnostics Report*

---

### Description

Print a diagnostics report, summarizing the state of a project using `renv`. This report can occasionally be useful when diagnosing issues with `renv`.

### Usage

```
diagnostics(project = NULL)
```

### Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

### Value

This function is normally called for its side effects.

## embed *Embed a Lockfile*

### Description

Use `embed()` to embed a compact representation of an `renv` lockfile directly within a file, using `use()` to automatically provision an R library when that script is run.

### Usage

```
embed(path = NULL, ..., lockfile = NULL, project = NULL)
```

### Arguments

| | |
|---|---|
| `path` | The path to an R or R Markdown script. |
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `lockfile` | The path to an `renv` lockfile. When `NULL` (the default), the project lockfile will be read (if any); otherwise, a new lockfile will be generated from the current library paths. |
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

### Details

Using `embed()` is useful if you'd like to be able to share "reproducible" R scripts – when these scripts are sourced, the generated call to `renv::use()` will ensure that an R library with the requested packages is automatically provisioned.

## equip *Install Required System Libraries*

### Description

Equip your system with libraries commonly-used during compilation of R packages. Currently only supported on Windows.

### Usage

```
equip()
```

### Value

This function is normally called for its side effects.

## Examples

```
## Not run:

# download useful build tools
renv::equip()


## End(Not run)
```

---

`history` *View Lockfile History*

---

## Description

Use your version control system to find prior versions of the `renv.lock` file that have been used in your project.

## Usage

```
history(project = NULL)
```

## Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Details

The `history()` function is currently only implemented for projects using `git` for version control.

## Value

An R `data.frame`, summarizing the commits in which `renv.lock` has been mutated.

## Examples

```
## Not run:

# get history of previous versions of renv.lock in VCS
db <- renv::history()

# choose an older commit
commit <- db$commit[5]

# revert to that version of the lockfile
renv::revert(commit = commit)


## End(Not run)
```

`hydrate` *Hydrate a Project*

## Description

Discover the R packages used within a project, and then install those packages into the active library. This effectively allows you to fork the state of your default R libraries for use within a project library.

## Usage

```
hydrate(
  packages = NULL,
  ...,
  library = NULL,
  update = FALSE,
  sources = NULL,
  project = NULL
)
```

## Arguments

| | |
|---|---|
| `packages` | The set of R packages to install. When `NULL`, the set of packages as reported by `dependencies()` is used. |
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `library` | The R library to be hydrated. When `NULL`, the active library as reported by `.libPaths()` is used. |
| `update` | Boolean; should `hydrate()` attempt to update already-installed packages if the requested package is already installed in the project library? Set this to `"all"` if you'd like *all* packages to be refreshed from the source library if possible. |
| `sources` | A set of library paths from which `renv` should attempt to draw packages. See **Sources** for more details. |
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Details

It may occasionally be useful to use `renv::hydrate()` to update the packages used within a project that has already been initialized. However, be warned that it's possible that the packages pulled in may not actually be compatible with the packages installed in the project library, so you should exercise caution when doing so.

## Value

A named R list, giving the packages that were used for hydration as well as the set of packages which were not found.

### Sources

hydrate() attempts to re-use packages already installed on your system, to avoid unnecessary attempts to download and install packages from remote sources. When NULL (the default), hydrate() will attempt to discover R packages from the following sources (in order):

- The user library,
- The site library,
- The system library,
- The renv cache.

If package is discovered in one of these locations, renv will attempt to copy or link that package into the requested library as appropriate.

### Missing Packages

If renv discovers that your project depends on R packages not currently installed in your user library, then it will attempt to install those packages from the active R repositories.

### Examples

```
## Not run:

# hydrate the active library
renv::hydrate()


## End(Not run)
```

---

## imbue *Imbue an renv Installation*

---

### Description

Imbue an renv installation into a project, thereby making the requested version of renv available within.

### Usage

```
imbue(project = NULL, version = NULL, quiet = FALSE)
```

### Arguments

| | |
|---|---|
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| version | The version of renv to install. If NULL, the version of renv currently installed will be used. The requested version of renv will be retrieved from the renv public GitHub repository, at https://github.com/rstudio/renv. |
| quiet | Boolean; avoid printing output during install of renv? |

### Details

Normally, this function does not need to be called directly by the user; it will be invoked as required by `init()` and `activate()`.

### Value

The project directory, invisibly. Note that this function is normally called for its side effects.

---

`init` *Initialize a Project*

---

### Description

Discover packages used within the current project, and then initialize a project-local private R library with those packages. The currently-installed versions of any packages in use (as detected within the default R libraries) are then installed to the project's private library.

### Usage

```
init(
  project = NULL,
  ...,
  profile = NULL,
  settings = NULL,
  bare = FALSE,
  force = FALSE,
  restart = interactive()
)
```

### Arguments

| | |
|---|---|
| `project` | The project directory. The R working directory will be changed to match the requested project directory. |
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `profile` | The profile to be activated. When `NULL`, the default profile is activated instead. See `vignette("profiles",package = "renv")` for more information. |
| `settings` | A list of settings to be used with the newly-initialized project. |
| `bare` | Boolean; initialize the project without attempting to discover and install R package dependencies? |
| `force` | Boolean; force initialization? By default, `renv` will refuse to initialize the home directory as a project, to defend against accidental mis-usages of `init()`. |
| `restart` | Boolean; attempt to restart the R session after initializing the project? A session restart will be attempted if the `"restart"` R option is set by the frontend embedding R. |

## Details

The primary steps taken when initializing a new project are:

1. R package dependencies are discovered within the R files used within the project with `dependencies()`;
2. Discovered packages are copied into the `renv` global package cache, so these packages can be re-used across future projects as necessary;
3. Any missing R package dependencies discovered are then installed into the project's private library;
4. A lockfile capturing the state of the project's library is created with `snapshot()`;
5. The project is activated with `activate()`.

If `renv` sees that the associated project has already been initialized and has a lockfile, then it will attempt to infer the appropriate action to take based on the presence of a private library. If no library is available, `renv` will restore the private library from the lockfile; if one is available, `renv` will ask if you want to perform a 'standard' init, restore from the lockfile, or activate the project without taking any further action.

## Value

The project directory, invisibly. Note that this function is normally called for its side effects.

## Infrastructure

`renv` will write or amend the following files in the project:

- `.Rprofile`: An auto-loader will be installed, so that new R sessions launched within the project are automatically loaded.
- `renv/activate.R`: This script is run by the previously-mentioned `.Rprofile` to load the project.
- `renv/.gitignore`: This is used to instruct Git to ignore the project's private library, as it should normally not be committed to a version control repository.
- `.Rbuildignore`: to ensure that the `renv` directory is ignored during package development; e.g. when attempting to build or install a package using `renv`.

## Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")
```

```
# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

---

`install` *Install Packages*

---

## Description

Install one or more R packages, from a variety of remote sources.

## Usage

```
install(
  packages = NULL,
  ...,
  library = NULL,
  type = NULL,
  rebuild = FALSE,
  prompt = interactive(),
  project = NULL
)
```

## Arguments

| | |
|---|---|
| `packages` | A character vector of R packages to install. Required package dependencies (`Depends`, `Imports`, `LinkingTo`) will be installed as required. |
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `library` | The R library to be used. When `NULL`, the active project library will be used instead. |
| `type` | The type of package to install ("source" or "binary"). Defaults to the value of `getOption("pkgType")`. |

| | |
|---|---|
| rebuild | Force packages to be rebuilt, thereby bypassing any installed versions of the package available in the cache? This can either be a boolean (indicating that all installed packages should be rebuilt), or a vector of package names indicating which packages should be rebuilt. |
| prompt | Boolean; prompt the user before taking any action? For backwards compatibility, `confirm` is accepted as an alias for `prompt`. |
| project | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Details

`install()` uses the same machinery as `restore()` when installing packages. In particular, this means that the local cache of package installations is used when possible. This helps to avoid re-downloading packages that have already been downloaded before, and re-compiling packages from source when a binary copy of that package is already available.

## Value

A named list of package records which were installed by `renv`.

## Project DESCRIPTION Files

If your project contains a `DESCRIPTION` file, then calling `install()` without any arguments will instruct `renv` to install the latest versions of all packages as declared within that `DESCRIPTION` file's `Depends`, `Imports` and `LinkingTo` fields; similar to how an R package might declare its dependencies.

If you have one or more packages that you'd like to install from a separate remote source, this can be accomplished by adding a Remotes: field to the `DESCRIPTION` file. See `vignette("dependencies",package = "devtools")` for more details. Alternatively, view the vignette online at https://devtools.r-lib.org/articles/dependencies.html.

Note that `install()` does not use the project's `renv.lock` when determining sources for packages to be installed. If you want to install packages using the sources declared in the lockfile, consider using `restore()` instead. Otherwise, you can declare the package sources in your `DESCRIPTION`'s Remotes: field.

## Remotes Syntax

`renv` supports a subset of the `remotes` syntax used for package installation, as described in https://remotes.r-lib.org/articles/dependencies.html. See the examples below for more details.

## Bioconductor

Packages from Bioconductor can be installed by using the bioc:: prefix. For example,

```
renv::install("bioc::Biobase")
```

will install the latest-available version of `Biobase` from Bioconductor.

`renv` depends on `BiocManager` (or, for older versions of R, `BiocInstaller`) for the installation of packages from Bioconductor. If these packages are not available, `renv` will attempt to automatically install them before fulfilling the installation request.

## Package Configuration

Many R packages have a `configure` script that needs to be run to prepare the package for installation. Arguments and environment variables can be passed through to those scripts in a manner similar to install.packages. In particular, the R options `configure.args` and `configure.vars` can be used to map package names to their appropriate configuration. For example:

```
# installation of RNetCDF may require us to set include paths for netcdf
configure.args = c(RNetCDF = "--with-netcdf-include=/usr/include/udunits2"))
options(configure.args = configure.args)
renv::install("RNetCDF")
```

This could also be specified as, for example,

```
options(
  configure.args.RNetCDF = "--with-netcdf-include=/usr/include/udunits2"
)
renv::install("RNetCDF")
```

Similarly, additional flags that should be passed to R CMD INSTALL can be set via the `install.opts` R option:

```
# installation of R packages using the Windows Subsystem for Linux
# may require the `--no-lock` flag to be set during install
options(install.opts = "--no-lock")
renv::install("xml2")
```

## Examples

```
## Not run:

# install the latest version of 'digest'
renv::install("digest")

# install an old version of 'digest' (using archives)
renv::install("digest@0.6.18")

# install 'digest' from GitHub (latest dev. version)
renv::install("eddelbuettel/digest")

# install a package from GitHub, using specific commit
renv::install("eddelbuettel/digest@df55b00bff33e945246eff2586717452e635032f")

# install a package from Bioconductor
# (note: requires the BiocManager package)
```

```
renv::install("bioc::Biobase")

# install a package, specifying path explicitly
renv::install("~/path/to/package")

# install packages as declared in the project DESCRIPTION file
renv::install()


## End(Not run)
```

---

`isolate` *Isolate a Project*

---

## Description

Copy packages from the `renv` cache directly into the project library, so that the project can continue to function independently of the `renv` cache.

## Usage

```
isolate(project = NULL)
```

## Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Details

After calling `isolate()`, `renv` will still be able to use the cache on future `install()`s and `restore()`s. If you'd prefer that `renv` copy packages from the cache, rather than use symlinks, you can set the `renv` configuration option:

```
options(renv.config.cache.symlinks = FALSE)
```

to force `renv` to copy packages from the cache, as opposed to symlinking them. If you'd like to disable the cache altogether for a project, you can use:

```
settings$use.cache(FALSE)
```

to explicitly disable the cache for the project.

## Value

The project directory, invisibly. Note that this function is normally called for its side effects.

## Examples

```
## Not run:

# isolate a project
renv::isolate()


## End(Not run)
```

---

load *Load a Project*

---

## Description

Load an renv project.

## Usage

```
load(project = getwd(), quiet = FALSE)
```

## Arguments

| | |
|---|---|
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| quiet | Boolean; be quiet during load? |

## Details

Calling renv::load() will set the session's library paths to use a project-local library, and perform some other work to ensure the project is properly isolated from other packages on the system.

Normally, renv::load() is called automatically by the project auto-loader written to the project .Rprofile by init(). This allows R sessions launched from the root of an renv project directory to automatically load that project, without requiring explicit action from the user. However, if preferred or necessary, one can call renv::load("<project>") to explicitly load an renv project located at a particular path.

Use activate() to activate (or re-activate) an renv project, so that newly-launched R sessions can automatically load the associated project.

## Value

The project directory, invisibly. Note that this function is normally called for its side effects.

## Examples

```
## Not run:

# load a project -- note that this is normally done automatically
# by the project's auto-loader, but calling this explicitly to
# load a particular project may be useful in some circumstances
renv::load()


## End(Not run)
```

---

`lockfile` *Programmatically Create and Modify a Lockfile*

---

## Description

This function provides an API for creating and modifying `renv` lockfiles. This can be useful when you'd like to programmatically generate or modify a lockfile – for example, because you want to update or change a package record in an existing lockfile.

## Usage

```
lockfile(file = NULL, project = NULL)
```

## Arguments

| | |
|---|---|
| `file` | The path to an existing lockfile. When no lockfile is provided, a new one will be created based on the current project context. If you want to create a blank lockfile, use `file = NA` instead. |
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## See Also

`lockfiles`, for a description of the structure of an `renv` lockfile.

## Examples

```
## Not run:

lock <- lockfile("renv.lock")

# set the repositories for a lockfile
lock$repos(CRAN = "https://cran.r-project.org")

# depend on digest 0.6.22
lock$add(digest = "digest@0.6.22")
```

```
# write to file
lock$write("renv.lock")


## End(Not run)
```

---

`lockfiles` *Lockfiles*

---

## Description

A **lockfile** records the state of a project at some point in time.

## Details

A lockfile captures the state of a project's library at some point in time. In particular, the package names, their versions, and their sources (when known) are recorded in the lockfile.

Projects can be restored from a lockfile using the `restore()` function. This implies reinstalling packages into the project's private library, as encoded within the lockfile.

While lockfiles are normally generated and used with `snapshot()` / `restore()`, they can also hand-edited if so desired. Lockfiles are written as `.json`, to allow for easy consumption by other tools.

An example lockfile follows:

```
{
  "R": {
    "Version": "3.6.1",
    "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": "0.7",
      "Source": "Repository",
```

```
      "Repository": "CRAN",
      "Hash": "908d95ccbfd1dd274073ef07a7c93934"
    }
  }
}
```

The sections used within a lockfile are described next.

**[renv]**

Information about the version of `renv` used to manage this project.

| | |
|---|---|
| **Version** | The version of the `renv` package used with this project. |

**[R]**

Properties related to the version of R associated with this project.

| | |
|---|---|
| **Version** | The version of R used. |
| **Repositories** | The R repositories used in this project. |

**[Packages]**

R package records, capturing the packages used or required by a project at the time when the lockfile was generated.

| | |
|---|---|
| **Package** | The package name. |
| **Version** | The package version. |
| **Source** | The location from which this package was retrieved. |
| **Repository** | The name of the repository (if any) from which this package was retrieved. |
| **Hash** | (Optional) A unique hash for this package, used for package caching. |

Additional remote fields, further describing how the package can be retrieved from its corresponding source, will also be included as appropriate (e.g. for packages installed from GitHub).

**[Python]**

Metadata related to the version of Python used with this project (if any).

| | |
|---|---|
| **Version** | The version of Python being used. |
| **Type** | The type of Python environment being used ("virtualenv", "conda", "system") |
| **Name** | The (optional) name of the environment being used. |

Note that the `Name` field may be empty. In that case, a project-local Python environment will be used instead (when not directly using a system copy of Python).

## See Also

Other reproducibility: `restore()`, `snapshot()`

---

`migrate` *Migrate a Project from Packrat to renv*

---

## Description

Migrate a project's infrastructure from Packrat to `renv`.

## Usage

```
migrate(
  project = NULL,
  packrat = c("lockfile", "sources", "library", "options", "cache")
)
```

## Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| `packrat` | Components of the Packrat project to migrate. See the default argument list for components of the Packrat project that can be migrated. Select a subset of those components for migration as appropriate. |

## Value

The project directory, invisibly. Note that this function is normally called for its side effects.

## Migration

When migrating Packrat projects to `renv`, the set of components migrated can be customized using the `packrat` argument. The set of components that can be migrated are as follows:

| Name | Description |
|---|---|
| `lockfile` | Migrate the Packrat lockfile (`packrat/packrat.lock`) to the `renv` lockfile (`renv.lock`). |
| `sources` | Migrate package sources from the `packrat/src` folder to the `renv` sources folder. Currently, only CRAN packag |
| `library` | Migrate installed packages from the Packrat library to the `renv` project library. |
| `options` | Migrate compatible Packrat options to the `renv` project. |
| `cache` | Migrate packages from the Packrat cache to the `renv` cache. |

## Examples

```
## Not run:

# migrate Packrat project infrastructure to renv
```

```
renv::migrate()


## End(Not run)
```

---

`modify` *Open the Lockfile for Editing*

---

## Description

Open a project's lockfile (if any) for editing. After edit, if the lockfile edited is associated with the active project, any state-related changes (e.g. to R repositories) will be updated in the current session.

## Usage

```
modify(project = NULL)
```

## Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Value

The project directory, invisibly. Note that this function is normally called for its side effects.

## Examples

```
## Not run:

# modify an existing lockfile
if (interactive())
  renv::modify()


## End(Not run)
```

paths *Path Customization*

## Description

Access the paths that renv uses for global state storage.

## Usage

```
paths
```

## Format

An object of class list of length 5.

## Details

By default, renv collects state into these folders:

| Platform | Location |
|---|---|
| Linux | ~/.local/share/renv |
| macOS | ~/Library/Application Support/renv |
| Windows | %LOCALAPPDATA%/renv |

For new installations of renv using R (>= 4.0.0), renv will use `tools::R_user_dir()` to resolve the root directory. If an renv root directory has already been created in one of the old locations, that will still be used. This change was made to comply with the CRAN policy requirements of R packages. By default, these paths resolve as:

| Platform | Location |
|---|---|
| Linux | ~/.cache/R/renv |
| macOS | ~/Library/Caches/org.R-project.R/R/renv |
| Windows | %LOCALAPPDATA%/R/cache/R/renv |

If desired, this path can be customized by setting the RENV_PATHS_ROOT environment variable. This can be useful if you'd like, for example, multiple users to be able to share a single global cache.

The various state sub-directories can also be individually adjusted, if so desired (e.g. you'd prefer to keep the cache of package installations on a separate volume). The various environment variables that can be set are enumerated below:

| Environment Variable | Description |
|---|---|
| RENV_PATHS_ROOT | The root path used for global state storage. |
| RENV_PATHS_LIBRARY | The path to the project library. |
| RENV_PATHS_LIBRARY_ROOT | The parent path for project libraries. |
| RENV_PATHS_LIBRARY_STAGING | The parent path used for staged package installs. |

| | |
|---|---|
| RENV_PATHS_LOCAL | The path containing local package sources. |
| RENV_PATHS_SOURCE | The path containing downloaded package sources. |
| RENV_PATHS_BINARY | The path containing downloaded package binaries. |
| RENV_PATHS_CACHE | The path containing cached package installations. |
| RENV_PATHS_PREFIX | An optional prefix to prepend to the constructed library / cache paths. |
| RENV_PATHS_RTOOLS | (Windows only) The path to Rtools. |
| RENV_PATHS_EXTSOFT | (Windows only) The path containing external software needed for compilation of Windows |
| RENV_PATHS_MRAN | The path containing MRAN-related metadata. See `vignette("mran", package = "renv")` |

Note that `renv` will append platform-specific and version-specific entries to the set paths as appropriate. For example, if you have set:

```
Sys.setenv(RENV_PATHS_CACHE = "/mnt/shared/renv/cache")
```

then the directory used for the cache will still depend on the `renv` cache version (e.g. `v2`), the R version (e.g. `3.5`) and the platform (e.g. `x86_64-pc-linux-gnu`). For example:

```
/mnt/shared/renv/cache/v2/R-3.5/x86_64-pc-linux-gnu
```

This ensures that you can set a single `RENV_PATHS_CACHE` environment variable globally without worry that it may cause collisions or errors if multiple versions of R needed to interact with the same cache.

If you need to share the same cache with multiple different Linux operating systems, you may want to set the `RENV_PATHS_PREFIX` environment variable to help disambiguate the paths used on Linux. For example, setting `RENV_PATHS_PREFIX = "ubuntu-bionic"` would instruct `renv` to construct a cache path like:

```
/mnt/shared/renv/cache/v2/ubuntu-bionic/R-3.5/x86_64-pc-linux-gnu
```

If this is required, it's strongly recommended that this environment variable is set in your R installation's `Renviron.site` file, typically located at `file.path(R.home("etc"),"Renviron.site")`, so that it can be active for any R sessions launched on that machine.

Starting from renv 0.13.0, you can also instruct `renv` to auto-generate an OS-specific component to include as part of library and cache paths, by setting the environment variable:

```
RENV_PATHS_PREFIX_AUTO = TRUE
```

The prefix will be constructed based on fields within the system's /etc/os-release file.

If reproducibility of a project is desired on a particular machine, it is highly recommended that the `renv` cache of installed packages + binary packages is backed up and persisted, so that packages can be easily restored in the future – installation of packages from source can often be arduous.

If you want these settings to persist in your project, it is recommended that you add these to an appropriate R startup file. For example, these could be set in:

- A project-local `.Renviron`;
- The user-level `.Renviron`;
- A file at `file.path(R.home("etc"),"Renviron.site")`.

Please see ?Startup for more details.

### Local Sources

If your project depends on one or R packages that are not available in any remote location, you can still provide a locally-available tarball for renv to use during restore. By default, these packages should be made available in the folder as specified by the RENV_PATHS_LOCAL environment variable. The package sources should be placed in a file at one of these locations:

- ${RENV_PATHS_LOCAL}/<package>_<version>.<ext>
- ${RENV_PATHS_LOCAL}/<package>/<package>_<version>.<ext>
- <project>/renv/local/<package>_<version>.<ext>
- <project>/renv/local/<package>/<package>_<version>.<ext>

where .<ext> is .tar.gz for source packages, or .tgz for binaries on macOS and .zip for binaries on Windows. During a restore(), packages installed from an unknown source will be searched for in this location.

### Projects

In order to determine whether a package can safely be removed from the cache, renv needs to know which projects are using packages from the cache. Since packages may be symlinked from the cache, and symlinks are by nature a one-way link, projects need to also report that they're using the renv cache.

To accomplish this, whenever renv is used with a project, it will record itself as being used within a file located at:

- ${RENV_PATHS_ROOT}/projects

This file is list of projects currently using the renv cache. With this, renv can crawl projects registered with renv and use that to determine if any packages within the cache are no longer in use, and can be removed.

### Examples

```
# get the path to the project library
path <- renv::paths$library()
```

---

## project *Retrieve the Active Project*

---

### Description

Retrieve the path to the active project (if any).

### Usage

```
project(default = NULL)
```

### Arguments

| | |
|---|---|
| default | The value to return when no project is currently active. Defaults to NULL. |

## Value

The active project directory, as a length-one character vector.

## Examples

```
## Not run:

# get the currently-active renv project
renv::project()


## End(Not run)
```

---

`purge` *Purge Packages from the Cache*

---

## Description

Purge packages from the cache. This can be useful if a package which had previously been installed in the cache has become corrupted or unusable, and needs to be reinstalled.

## Usage

```
purge(package, ..., version = NULL, hash = NULL, prompt = interactive())
```

## Arguments

| | |
|---|---|
| `package` | A single package to be removed from the cache. |
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `version` | The package version to be removed. When `NULL`, all versions of the requested package will be removed. |
| `hash` | The specific hashes to be removed. When `NULL`, all hashes associated with a particular package's version will be removed. |
| `prompt` | Boolean; prompt the user before taking any action? For backwards compatibility, `confirm` is accepted as an alias for `prompt`. |

## Details

`purge()` is an inherently destructive option. It removes packages from the cache, and so any project which had symlinked that package into its own project library would find that package now unavailable. These projects would hence need to reinstall any purged packages. Take heed of this in case you're looking to purge the cache of a package which is difficult to install, or if the original sources for that package are no longer available!

## Value

The set of packages removed from the renv global cache, as a character vector of file paths.

## Examples

```
## Not run:

# remove all versions of 'digest' from the cache
renv::purge("digest")

# remove only a particular version of 'digest' from the cache
renv::purge("digest", version = "0.6.19")


## End(Not run)
```

---

# rebuild *Rebuild the Packages in your Project Library*

---

## Description

Rebuild and reinstall packages in your library. This can be useful as a diagnostic tool – for example, if you find that one or more of your packages fail to load, and you want to ensure that you are starting from a clean slate.

## Usage

```
rebuild(
  packages = NULL,
  recursive = TRUE,
  ...,
  type = NULL,
  prompt = interactive(),
  library = NULL,
  project = NULL
)
```

## Arguments

| | |
|---|---|
| packages | The package(s) to be rebuilt. When NULL, all packages in the library will be reinstalled. |
| recursive | Boolean; should dependencies of packages be rebuilt recursively? Defaults to TRUE. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |
| type | The type of package to install ("source" or "binary"). Defaults to the value of getOption("pkgType"). |

| | |
|---|---|
| prompt | Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt. |
| library | The R library to be used. When NULL, the active project library will be used instead. |
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Value

A named list of package records which were installed by renv.

## Examples

```
## Not run:

# rebuild the 'dplyr' package + all of its dependencies
renv::rebuild("dplyr", recursive = TRUE)

# rebuild only 'dplyr'
renv::rebuild("dplyr", recursive = FALSE)


## End(Not run)
```

---

# record *Update Package Records in a Lockfile*

---

## Description

Use record() to record a new entry within an existing renv lockfile.

## Usage

```
record(records, lockfile = NULL, project = NULL)
```

## Arguments

| | |
|---|---|
| records | A list of named records, mapping package names to a definition of their source. See **Records** for more details. |
| lockfile | The path to a lockfile. By default, the project lockfile is used. |
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Details

This function can be useful when you need to change one or more of the package records within an renv lockfile – for example, because a recorded package cannot be restored in a particular environment, and you know of a suitable alternative.

## Records

Records can be provided either using the **remotes** short-hand syntax, or by using an R list of entries to record within the lockfile. See `?lockfiles` for more information on the structure of a package record.

## Examples

```
## Not run:

# use digest 0.6.22 from package repositories -- different ways
# of specifying the remote. use whichever is most natural
renv::record("digest@0.6.22")
renv::record(list(digest = "0.6.22"))
renv::record(list(digest = "digest@0.6.22"))

# alternatively, provide a full record as a list
digest_record <- list(
  Package = "digest",
  Version = "0.6.22",
  Source  = "Repository",
  Repository = "CRAN"
)

renv::record(list(digest = digest_record))


## End(Not run)
```

---

`refresh` *Refresh the Local Cache of Available Packages*

---

## Description

Query the active R package repositories for available packages, and update the in-memory cache of those packages.

## Usage

```
refresh()
```

## Details

Note that R also maintains its own on-disk cache of available packages, which is used by `available.packages()`. Calling `refresh()` will force an update of both types of caches. `renv` prefers using an in-memory cache as on occasion the temporary directory can be slow to access (e.g. when it is a mounted network filesystem).

## Value

A list of package databases, invisibly – one for each repository currently active in the R session. Note that this function is normally called for its side effects.

## Examples

```
## Not run:

# check available packages
db <- available.packages()

# wait some time (suppose packages are uploaded / changed in this time)
Sys.sleep(5)

# refresh the local available packages database
# (the old locally cached db will be removed)
db <- renv::refresh()


## End(Not run)
```

---

rehash *Re-Hash Packages in the renv Cache*

---

## Description

Re-hash packages in the renv cache, ensuring that any previously-cached packages are copied to a new cache location appropriate for this version of renv. This can be useful if the cache scheme has changed in a new version of renv, but you'd like to preserve your previously-cached packages.

## Usage

```
rehash(prompt = interactive(), ...)
```

## Arguments

| | |
|---|---|
| prompt | Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |

## Details

Any packages which are re-hashed will retain links to the location of the newly-hashed package, ensuring that prior installations of renv can still function as expected.

## remote *Resolve a Remote*

### Description

Given a remote specification, resolve it into an `renv` package record that can be used for download and installation (e.g. with install).

### Usage

```
remote(spec)
```

### Arguments

| | |
|---|---|
| `spec` | A remote specification. |

## remove *Remove Packages*

### Description

Remove (uninstall) R packages.

### Usage

```
remove(packages, ..., library = NULL, project = NULL)
```

### Arguments

| | |
|---|---|
| `packages` | A character vector of R packages to remove. |
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `library` | The library from which packages should be removed. When `NULL`, the active library (that is, the first entry reported in `.libPaths()`) is used instead. |
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

### Value

A vector of package records, describing the packages (if any) which were successfully removed.

## Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

---

# renv_lockfile_from_manifest

*Generate* renv.lock *from an RStudio Connect* manifest.json

---

## Description

Use renv_lockfile_from_manifest() to convert a manifest.json file from an RStudio Connect content bundle into an renv.lock lockfile.

## Usage

```
renv_lockfile_from_manifest(manifest, lockfile = NA)
```

## Arguments

| | |
|---|---|
| manifest | The path to a manifest.json file. |

`lockfile` The path to the lockfile to be generated and / or updated. When `NA` (the default), the generated lockfile is returned as an R object; otherwise, the lockfile will be written to the path specified by `lockfile`.

## Details

This function can be useful when you need to recreate the package environment of a piece of content that is deployed to RStudio Connect. The content bundle contains a `manifest.json` file that is used to recreate the package environment. This function will let you convert that manifest file to an `renv.lock` file. Run `renv::restore()` after you've converted the file to restore the package environment.

By default the `lockfile` argument is set to `NA`. This will not create a new `renv.lock` file. Rather, it will return a lockfile object (see `?lockfile`) that can be used to create a new `renv.lock` file. If `lockfile` is set to a character string, a new file will be created with that path – e.g. `renv.lock` – and the lockfile object will be returned.

## Value

An `renv` lockfile.

---

# `restore` *Restore a Project*

---

## Description

Restore a project's dependencies from a lockfile, as previously generated by `snapshot()`.

## Usage

```
restore(
  project = NULL,
  ...,
  library = NULL,
  lockfile = NULL,
  packages = NULL,
  exclude = NULL,
  rebuild = FALSE,
  repos = NULL,
  clean = FALSE,
  prompt = interactive()
)
```

## Arguments

`project` The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.

| | |
|---|---|
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `library` | The library paths to be used during restore. See **Library** for details. |
| `lockfile` | The lockfile to be used for restoration of the associated project. When `NULL`, the most recently generated lockfile for this project is used. |
| `packages` | A subset of packages recorded in the lockfile to restore. When `NULL` (the default), all packages available in the lockfile will be restored. Any required recursive dependencies of the requested packages will be restored as well. |
| `exclude` | A subset of packages to be excluded during restore. This can be useful for when you'd like to restore all but a subset of packages from a lockfile. Note that if you attempt to exclude a package which is required as the recursive dependency of another package, your request will be ignored. |
| `rebuild` | Force packages to be rebuilt, thereby bypassing any installed versions of the package available in the cache? This can either be a boolean (indicating that all installed packages should be rebuilt), or a vector of package names indicating which packages should be rebuilt. |
| `repos` | The repositories to use during restore, for packages installed from CRAN or another similar R package repository. When set, this will override any repositories declared in the lockfile. See also the `repos.override` option in config for an alternate way to provide a repository override. |
| `clean` | Boolean; remove packages not recorded in the lockfile from the target library? Use `clean = TRUE` if you'd like the library state to exactly reflect the lockfile contents after `restore()`. |
| `prompt` | Boolean; prompt the user before taking any action? For backwards compatibility, `confirm` is accepted as an alias for `prompt`. |

## Value

A named list of package records which were installed by `renv`.

## Package Repositories

By default, the package repositories encoded in the lockfile will be used during restore, as opposed to the repositories that might already be set in the current session (through `getOption("repos")`). If you'd like to override the repositories used by `renv` during restore, you can use, for example:

```
renv::restore(repos = c(CRAN = <...>))
```

See also the `repos.override` option in config for an alternate way to provide a repository override.

## Library

When `renv::restore()` is called, packages from the lockfile are compared against packages currently installed in the library paths specified by `library`. Any packages which have changed will then be installed into the default library. If `clean = TRUE`, then packages that exist within the default library, but aren't recorded in the lockfile, will be removed as well.

## See Also

Other reproducibility: lockfiles, snapshot()

## Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

---

# revert *Revert Lockfile*

---

## Description

Revert the lockfile to its contents at a prior commit.

## Usage

```
revert(commit = "HEAD", ..., project = NULL)
```

### Arguments

| | |
|---|---|
| commit | The commit associated with a prior version of the lockfile. |
| ... | Optional arguments; currently unused. |
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

### Details

The revert() function is currently only implemented for projects using git for version control.

### Value

The commit used when reverting renv.lock. Note that this function is normally called for its side effects.

### Examples

```
## Not run:

# get history of previous versions of renv.lock in VCS
db <- renv::history()

# choose an older commit
commit <- db$commit[5]

# revert to that version of the lockfile
renv::revert(commit = commit)


## End(Not run)
```

---

## run *Run a Script*

---

### Description

Run an R script, in the context of a project using renv. The script will be run within an R subprocess.

### Usage

```
run(script, ..., job = NULL, name = NULL, project = NULL)
```

### Arguments

| | |
|---|---|
| script | The path to an R script. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |
| job | Run the requested script as an RStudio job? Requires a recent version of both RStudio and the rstudioapi packages. When NULL, the script will be run as a job if possible, and as a regular R process launched by system2() if not. |
| name | The name to associate with the job, for scripts run as a job. |
| project | The path to the renv project. This project will be loaded before the requested script is executed. When NULL (the default), renv will automatically determine the project root for the associated script if possible. |

### Value

The project directory, invisibly. Note that this function is normally called for its side effects.

---

## scaffold *Generate renv Project Infrastructure*

---

### Description

Write the renv project infrastructure for a project.

### Usage

```
scaffold(
  project = NULL,
  version = NULL,
  repos = getOption("repos"),
  settings = NULL
)
```

### Arguments

| | |
|---|---|
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| version | The version of renv to associate with this project. By default, the version of renv currently installed is used. |
| repos | The R repositories to associate with this project. |
| settings | A list of renv settings, to be applied to the project after creation. These should map setting names to the desired values. See settings for more details. |

### Details

Invoking `renv::scaffold()` will:

- Install `renv` into the project library,
- Update the project `.Rprofile` so that `renv` is automatically loaded for new R sessions launched in this project, and
- Write a bare lockfile `renv.lock`.

### Examples

```
## Not run:
# create scaffolding with 'devtools' ignored
renv::scaffold(settings = list(ignored.packages = "devtools"))

## End(Not run)
```

---

## `settings` *Project Settings*

---

### Description

Define project-local settings that can be used to adjust the behavior of `renv` with your particular project.

### Usage

```
settings
```

### Format

An object of class `list` of length 8.

### Settings

`external.libraries` A vector of library paths, to be used in addition to the project's own private library. This can be useful if you have a package available for use in some global library, but for some reason `renv` is not able to install that package (e.g. sources or binaries for that package are not publicly available, or you have been unable to orchestrate the pre-requisites for installing some packages from source on your machine).

`ignored.packages` A vector of packages, which should be ignored when attempting to snapshot the project's private library. Note that if a package has already been added to the lockfile, that entry in the lockfile will not be ignored.

`package.dependency.fields` During dependency discovery, `renv` uses the fields of an installed package's `DESCRIPTION` file to determine that package's recursive dependencies. By default, the `Imports`, `Depends` and `LinkingTo` fields are used. If you'd prefer that `renv` also captures the `Suggests` dependencies for a package, you can set this to `c("Imports","Depends","LinkingTo","Suggests")`.

`r.version` The version of R to encode within the lockfile. This can be set as a project-specific option if you'd like to allow multiple users to use the same `renv` project with different versions of R. `renv` will still warn the user if the major + minor version of R used in a project does not match what is encoded in the lockfile.

`snapshot.type` The type of snapshot to perform by default. See snapshot for more details.

`use.cache` Use a global cache of R packages. When active, `renv` will install packages into a global cache, and link packages from the cache into your `renv` projects as appropriate. This can greatly save on disk space and install time when for R packages which are used across multiple projects in the same environment.

`vcs.ignore.library` Set whether the `renv` project library is excluded from version control.

`vcs.ignore.local` Set whether `renv` project-specific local sources are excluded from version control.

## Defaults

You can change the default values of these settings for newly-created `renv` projects by setting R options for `renv.settings` or renv.settings.<name>. For example:

```
options(renv.settings = list(snapshot.type = "all"))
options(renv.settings.snapshot.type = "all")
```

If both of the `renv.settings` and renv.settings.<name> options are set for a particular key, the option associated with renv.settings.<name> is used instead. We recommend setting these in an appropriate startup profile, e.g. ~/.Rprofile or similar.

## Examples

```
## Not run:

# view currently-ignored packaged
renv::settings$ignored.packages()

# ignore a set of packages
renv::settings$ignored.packages("devtools", persist = FALSE)


## End(Not run)
```

---

`snapshot` *Snapshot a Project*

---

## Description

Call `snapshot()` to create a **lockfile** capturing the state of a project's R package dependencies. The lockfile can be used to later restore these project's dependencies as required.

## Usage

```
snapshot(
  project = NULL,
  ...,
  library = NULL,
  lockfile = paths$lockfile(project = project),
  type = settings$snapshot.type(project = project),
  packages = NULL,
  prompt = interactive(),
  force = FALSE,
  reprex = FALSE
)
```

## Arguments

| | |
|---|---|
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |
| library | The R libraries to snapshot. When NULL, the active R libraries (as reported by .libPaths()) are used. |
| lockfile | The location where the generated lockfile should be written. By default, the lockfile is written to a file called renv.lock in the project directory. When NULL, the lockfile (as an R object) is returned directly instead. |
| type | The type of snapshot to perform. See **Snapshot Type** for more details. |
| packages | A vector of packages to be included in the lockfile. When NULL (the default), all packages relevant for the type of snapshot being performed will be included. When set, the type argument is ignored. Recursive dependencies of the specified packages will be added to the lockfile as well. |
| prompt | Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt. |
| force | Boolean; force generation of a lockfile even when pre-flight validation checks have failed? |
| reprex | Boolean; generate output appropriate for embedding the lockfile as part of a reprex? |

## Details

See the lockfile documentation for more details on the structure of a lockfile.

## Value

The generated lockfile, as an R object (invisibly). Note that this function is normally called for its side effects.

## Snapshot Type

Depending on how you prefer to manage dependencies, you might prefer selecting a different snapshot mode. The modes available are as follows:

`"all"` Capture all packages within the active R libraries in the lockfile. This is the quickest and simplest method, but may lead to undesired packages (e.g. development dependencies) entering the lockfile.

`"implicit"` Only capture packages which appear to be used in your project in the lockfile. The intersection of packages installed in your R libraries, alongside those used in your R code as inferred by `renv::dependencies()`, will enter the lockfile. This helps ensure that only the packages your project requires will enter the lockfile, but may be slower if your project contains a large number of files. If this becomes an issue, you might consider using `.renvignore` files to limit which files `renv` uses for dependency discovery, or explicitly declaring your required dependencies in a `DESCRIPTION` file. You can also force a dependency on a particular package by writing e.g. library(<package>) into a file called `dependencies.R`.

`"explicit"` Only capture packages which are explicitly listed in the project `DESCRIPTION` file. This workflow is recommended for users who wish to manage their project's R package dependencies directly.

`"custom"` Like `"implicit"`, but use a custom user-defined filter instead. The filter should be specified by the R option `renv.snapshot.filter`, and should either be a character vector naming a function (e.g. `"package::method"`), or be a function itself. The function should only accept one argument (the project directory), and should return a vector of package names to include in the lockfile.

By default, `"implicit"`-style snapshots are used. The snapshot type can be configured on a project-specific basis using the `renv` project settings mechanism. For example, to use `"explicit"` snapshots in a project:

```
renv::settings$snapshot.type("explicit")
```

When the `packages` argument is set, `type` is ignored, and instead only the requested set of packages, and their recursive dependencies, will be written to the lockfile.

## See Also

Other reproducibility: `lockfiles`, `restore()`

## Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)
```

```
# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

---

`status` *Status*

---

## Description

Report differences between the project's lockfile and the current state of the project's library (if any).

## Usage

```
status(project = NULL, ..., library = NULL, lockfile = NULL, cache = FALSE)
```

## Arguments

| | |
|---|---|
| `project` | The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| `...` | Unused arguments, reserved for future expansion. If any arguments are matched to `...`, `renv` will signal an error. |
| `library` | The library paths. By default, the library paths associated with the requested project are used. |
| `lockfile` | The path to a lockfile. By default, the project lockfile (called `renv.lock`) is used. |
| `cache` | Boolean; perform diagnostics on the global package cache? When `TRUE`, `renv` will validate that the packages installed into the cache are installed at the expected + proper locations, and validate the hashes used for those storage locations. |

## Value

This function is normally called for its side effects.

## Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

---

`update` *Update Packages*

---

## Description

Update packages which are currently out-of-date. Currently, only CRAN and GitHub package sources are supported.

## Usage

```
update(
  packages = NULL,
  ...,
```

```
  exclude = NULL,
  library = NULL,
  rebuild = FALSE,
  check = FALSE,
  prompt = interactive(),
  project = NULL
)
```

## Arguments

| | |
|---|---|
| packages | A character vector of R packages to update. When NULL (the default), all packages will be updated. |
| ... | Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error. |
| exclude | A set of packages to explicitly exclude from updating. Use renv::update(exclude = <...>) to update all packages except for a specific set of excluded packages. |
| library | The R library to be used. When NULL, the active project library will be used instead. |
| rebuild | Force packages to be rebuilt, thereby bypassing any installed versions of the package available in the cache? This can either be a boolean (indicating that all installed packages should be rebuilt), or a vector of package names indicating which packages should be rebuilt. |
| check | Boolean; check for package updates without actually installing available updates? This is useful when you'd like to determine what updates are available, without actually installing those updates. |
| prompt | Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt. |
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Details

Updates will only be checked from the same source – for example, if a package was installed from GitHub, but a newer version is available on CRAN, that updated version will not be seen.

You can call renv::update() with no arguments to update all packages within the project, excluding any packages ignored via the ignored.packages project setting. Use the exclude argument to further refine the exclusion criteria if desired.

## Value

A named list of package records which were installed by renv.

## Examples

```
## Not run:

# update the 'dplyr' package
```

```
renv::update("dplyr")


## End(Not run)
```

---

upgrade *Upgrade renv*

---

## Description

Upgrade the version of renv associated with a project.

## Usage

```
upgrade(project = NULL, version = NULL, reload = NULL, prompt = interactive())
```

## Arguments

| | |
|---|---|
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |
| version | The version of renv to be installed. By default, the latest version of renv as available on the active R package repositories is used. |
| reload | Boolean; reload renv after install? When NULL (the default), renv will be reloaded only if updating renv for the active project. Note that this may fail if you've loaded packages which also depend on renv. |
| prompt | Boolean; prompt upgrade before proceeding? |

## Details

By default, this function will attempt to install the latest version of renv as available on the active R package repositories. If you'd instead like to try out a development version of renv, you can explicitly request a different version of renv and that version of the package will be downloaded and installed from GitHub. Use version = "master" to install the latest development version of renv, as from the renv project's GitHub page.

## Value

A boolean value, indicating whether the requested version of renv was successfully installed. Note that this function is normally called for its side effects.

## Examples

```
## Not run:

# upgrade to the latest version of renv
renv::upgrade()

# upgrade to the latest version of renv on GitHub (development version)
```

```
renv::upgrade(version = "master")


## End(Not run)
```

---

use *Use a set of Packages*

---

## Description

Given a set of R package requirements, install those packages into the library path requested via `library`, and then activate that library path.

## Usage

```
use(
  ...,
  lockfile = NULL,
  library = NULL,
  isolate = FALSE,
  attach = FALSE,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| `...` | The R packages to be used with this script. Ignored if `lockfile` is non-`NULL`. |
| `lockfile` | The lockfile to use. When supplied, `renv` will use the packages as declared in the lockfile. |
| `library` | The library path into which the requested packages should be installed. When `NULL` (the default), a library path within the R temporary directory will be generated and used. Note that this same library path will be re-used on future calls to `renv::use()`, allowing `renv::use()` to be used multiple times within a single script. |
| `isolate` | Boolean; should the active library paths be included in the set of library paths activated for this script? Set this to `TRUE` if you only want the packages provided to `renv::use()` to be visible on the library paths. |
| `attach` | Boolean; should the set of requested packages be automatically attached? If `TRUE`, packages will be loaded and attached via a call to `library()` after install. Ignored if `lockfile` is non-`NULL`. |
| `verbose` | Boolean; be verbose while installing packages? |

### Details

renv::use() is intended to be used within standalone R scripts. It can be useful when you'd like to specify an R script's dependencies directly within that script, and have those packages automatically installed and loaded when the associated script is run. In this way, an R script can more easily be shared and re-run with the exact package versions requested via use().

renv::use() is inspired in part by the groundhog package, which provides an alternate mechanism for specifying a script's R package requirements within that same R script.

### Value

This function is normally called for its side effects.

---

## use_python *Use Python*

---

### Description

Associate a version of Python with your project.

### Usage

```
use_python(
  python = NULL,
  ...,
  type = c("auto", "virtualenv", "conda", "system"),
  name = NULL,
  project = NULL
)
```

### Arguments

| | |
|---|---|
| python | The path to the version of Python to be used with this project. See **Finding Python** for more details. |
| ... | Optional arguments; currently unused. |
| type | The type of Python environment to use. When "auto" (the default), virtual environments will be used. |
| name | The name or path that should be used for the associated Python environment. If NULL and python points to a Python executable living within a pre-existing virtual environment, that environment will be used. Otherwise, a project-local environment will be created instead, using a name generated from the associated version of Python. |
| project | The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead. |

## Details

When Python integration is active, renv will:

- Save metadata about the requested version of Python in renv.lock – in particular, the Python version, and the Python type ("virtualenv", "conda", "system"),
- Capture the set of installed Python packages during renv::snapshot(),
- Re-install the set of recorded Python packages during renv::restore().

In addition, when the project is loaded, the following actions will be taken:

- The RENV_PYTHON environment variable will be set, indicating the version of Python currently active for this sessions,
- The RETICULATE_PYTHON environment variable will be set, so that the reticulate package can automatically use the requested copy of Python as appropriate,
- The requested version of Python will be placed on the PATH, so that attempts to invoke Python will resolve to the expected version of Python.

You can override the version of Python used in a particular project by setting the RENV_PYTHON environment variable; e.g. as part of the project's .Renviron file. This can be useful if you find that renv is unable to automatically discover a compatible version of Python to be used in the project.

## Value

TRUE, indicating that the requested version of Python has been successfully activated. Note that this function is normally called for its side effects.

## Finding Python

In interactive sessions, when python = NULL, renv will prompt for an appropriate version of Python. renv will search a pre-defined set of locations when attempting to find Python installations on the system:

- getOption("renv.python.root"),
- /opt/python,
- /opt/local/python,
- ~/opt/python,
- /usr/local/opt (for macOS Homebrew-installed copies of Python),
- /opt/homebrew/opt (for M1 macOS Homebrew-installed copies of Python),
- ~/.pyenv/versions,
- Python instances available on the PATH.

In non-interactive sessions, renv will first check the RETICULATE_PYTHON environment variable; if that is unset, renv will look for Python on the PATH. It is recommended that the version of Python to be used is explicitly supplied for non-interactive usages of use_python().

## Warning

We strongly recommend using Python virtual environments, for a few reasons:

1. If something goes wrong with a local virtual environment, you can safely delete that virtual environment, and then re-initialize it later, without worry that doing so might impact other software on your system.
2. If you choose to use a "system" installation of Python, then any packages you install or upgrade will be visible to any other application that wants to use that same Python installation. Using a virtual environment ensures that any changes made are isolated to that environment only.
3. Choosing to use Anaconda will likely invite extra frustration in the future, as you may be required to upgrade and manage your Anaconda installation as new versions of Anaconda are released. In addition, Anaconda installations tend to work poorly with software not specifically installed as part of that same Anaconda installation.

In other words, we recommend selecting "system" or "conda" only if you are an expert Python user who is already accustomed to managing Python / Anaconda installations on your own.

## Examples

```
## Not run:

# use python with a project
renv::use_python()

# use python with a project; create the environment
# within the project directory in the '.venv' folder
renv::use_python(name = ".venv")

# use python with a pre-existing virtual environment located elsewhere
renv::use_python(name = "~/.virtualenvs/env")

# use virtualenv python with a project
renv::use_python(type = "virtualenv")

# use conda python with a project
renv::use_python(type = "conda")


## End(Not run)
```

# Index