

Package: rpix (via r-universe)

October 20, 2026

Type Package

Title Manage R Project Dependencies with 'Pixi'

Version 0.9.1

Description An interface to the 'Pixi' package manager (<<https://pixi.prefix.dev>>) from R. Add, remove and search for 'conda' packages, and set up and work with 'Pixi' projects from the R console. If 'Pixi' isn't installed, 'rpix' can install it.

License MIT + file LICENSE

Encoding UTF-8

Language en-GB

Config/testthat/edition 3

URL <https://roald-arboel.com/rpix/>, <https://github.com/roaldarbol/rpix>

BugReports <https://github.com/roaldarbol/rpix/issues>

Roxygen list(markdown = TRUE)

SystemRequirements pixi (>= 0.77)

Imports askpass, cli, jsonlite, processx, rlang

Suggests knitr, rcmdcheck, rmarkdown, testthat (>= 3.0.0), withr

Config/Needs/website pkgdown, quarto

Config/roxygen2/version 8.1.0

Repository <https://roaldarbol.r-universe.dev>

Date/Publication 2026-10-20 22:07:08 UTC

RemoteUrl <https://github.com/roaldarbol/rpix>

RemoteRef HEAD

RemoteSha e1ecd258613a1166f4afcc00f19e246146c96b48

Contents

install_pixi	2
pixi_activate	3
pixi_add	4
pixi_add_channel	7
pixi_auth_login	8
pixi_check_matrix	9
pixi_environments	11
pixi_import_description	13
pixi_import_renv	14
pixi_info	16
pixi_install	17
pixi_r	19
pixi_remove	20
pixi_scan	21
pixi_search	22
pixi_set_date	23
pixi_sitrep	24
pixi_switch	25
pixi_tasks	26
use_pixi	28
use_pixi_direnv	30
use_pixi_github_action	32
use_pixi_ide	33
use_pixi_jupyter	35
use_pixi_python	36
Index	37

install_pixi	<i>Install Pixi</i>
--------------	---------------------

Description

Install Pixi from R, with Pixi's official installer, so you don't need a terminal. It installs Pixi into `~/.pixi/bin` (or `$PIXI_HOME/bin`), where `rpix` finds it straight away. When an `rpix` function can't find Pixi, it offers to run `install_pixi()`.

By default, the installer also adds Pixi to your shell's `PATH`, so `pixi` works in a terminal too, after you open a new one. If Pixi is installed already, nothing happens unless `force = TRUE`; update an existing Pixi with `pixi self-update` in a terminal.

The installer is Pixi's own, `install.sh` (or `install.ps1` on Windows) from `pixi.sh`. It may send `prefix.dev` an anonymous install count; set `PIXI_NO_TELEMETRY=1` to turn that off. See <https://pixi.prefix.dev/latest/installation/>.

Usage

```
install_pixi(version = NULL, update_path = TRUE, force = FALSE)
```

Arguments

version	Optional. The version of Pixi, such as "0.81.0". Defaults to the latest.
update_path	If TRUE, let the installer add Pixi to your shell's PATH, e.g. in ~/.zshrc. If FALSE, only rpix will find it.
force	If TRUE, install even if Pixi is installed already.

Value

The path to Pixi, invisibly.

Examples

```
## Not run:
install_pixi()

## End(Not run)
```

`pixi_activate`

Activate the project's Pixi environment in the running R

Description

Make R that was started directly from a Pixi environment, as IDEs such as Positron and VS Code do, behave like R started with `pixi run R`. Called from the project's `.Rprofile`, which `use_pixi()` sets up.

- If Pixi didn't activate the environment, its environment variables are set, as `pixi shell-hook` reports them. Some packages need them, e.g. to find their data.
- Your personal library is removed from `.libPaths()`, and the project's own library added if it exists. Other libraries, such as temporary ones that devtools adds, are kept.
- `install.packages()` adds packages with `pixi_add()` instead, so they're recorded in `pixi.toml` and `pixi.lock`. For a CRAN package that isn't on conda-forge, `pixi_add()` offers to build it from source. For a package already in `pixi.toml`, it says to update it with `pixi_update()` instead. `repos` is ignored, and other arguments Pixi can't use, such as `lib`, give an error. Package files and folders are installed with `utils::install.packages()`, so `pixi.toml` doesn't record them. Turn this off with `options(rpix.install_packages = FALSE)`.
- In a package, in an interactive session, `rpix` notices when `DESCRIPTION` comes to need packages that `pixi.toml` doesn't have, e.g. after `usethis::use_package()` or an edit, and offers to add them with `pixi_import_description()`. It also says when `DESCRIPTION` no longer lists a package, which you can remove with `pixi_remove()`. Turn this off with `options(rpix.watch_description = FALSE)`.
- If R isn't the R of a Pixi environment in this project, nothing changes; in an interactive session you get a hint instead. In RStudio, it also offers to reopen the project in a new RStudio with the environment's R, and to close this one. Turn that off with `RPIX_OFFER_SWITCH=false` in your `.Renviron`.

- If the project also has renv, e.g. because RStudio's New Project set it up, it says so: renv and Pixi both manage the project's packages.
- If the project was moved or copied, Pixi's activation cache, which has the old location, is removed, and environments that need reinstalling are named.

It never fails: problems are reported as messages.

Usage

```
pixi_activate(path = getwd(), quiet = !interactive())
```

Arguments

path	The project. Defaults to the working directory, which is where R reads .Rprofile from.
quiet	If TRUE, show no messages.

Value

A list, invisibly: `activated` is TRUE if environment variables were set, and `lib_paths` the new library paths, or NULL if they weren't changed.

Examples

```
## Not run:
pixi_activate()

## End(Not run)
```

pixi_add	<i>Add packages</i>
----------	---------------------

Description

Add packages to the Pixi manifest and install them. Pixi only adds them if they can be solved together with the rest of the project's dependencies.

R package names are translated to conda names: "dplyr" becomes r-dplyr and "Rcpp" becomes r-rcpp (conda names are lowercase). Use a prefix to say where a package comes from:

- "bioc::DESeq2": a Bioconductor package (bioconductor-deseq2 from the bioconda channel, which is added to the project if needed). Without the prefix, a package that isn't on conda-forge is looked for on bioconda too, so "DESeq2" works as well. Bioconductor's annotation and data packages, such as org.Hs.eg.db, download their data with an install script, which Pixi only runs in projects that allow it; `pixi_add()` offers to allow it.
- "conda::gdal": a conda package that isn't an R package, used as is. Names containing - or _, such as "c-compiler", are also used as is.
- "cran::dplyr": same as "dplyr".

- "pypi::pydytuesday" (experimental): a Python package from PyPI, added with `pixi add --pypi`, for Python in the project (see `use_pixi_python()`).
- "github::user/repo" (experimental): an R package on GitHub, which Pixi builds from source with its R build backend, `pixi-build-r`. Add `@ref` for a branch, tag or commit, as in "github::cran/praise@1.0.0"; `pixi.lock` records the exact commit either way. `rpix` writes it into `pixi.toml`, turns on Pixi's `pixi-build` preview, and pins the build to the project's R. Its dependencies come from conda-forge. It asks for `pixi-build-r` 0.2.0 or later, which adds the Rust, CMake, `pkg-config` and R's recommended packages a package needs; a project with packages as of a date before 2026-10-15 can't get it (see `pixi_set_date()`). `rpix` also adds what `pixi-build-r` doesn't: Java when `SystemRequirements` names it, and in a project for Windows, `MSYS2`'s `make`, `sed` and `coreutils`, and `MinGW`'s compilers in place of `pixi-build-r`'s. See <https://pixi.prefix.dev/latest/build/backends/pixi-build-r/>. This may change, e.g. to use `pixi add` once it can set a build backend; see <https://github.com/roaldarbol/rpix/issues/96>. While Pixi builds, `rpix` shows a line rather than the build's log, unless the build fails.

In a project with several environments, Pixi doesn't install after adding or removing, so `pixi_add()` and `pixi_remove()` install the environment R runs in (or else the default one), when the change applies to it.

When it adds R (`r-base`) to the default feature, it also adds `conda-ecosystem-user-package-isolation`, unless it's there, to keep your personal R library out of the environment's R, as `use_pixi()` does.

Pixi's output is shown once it has added the packages, or if it fails in a way `rpix` can't handle, but not when `rpix` can, e.g. by looking on bioconda. To see Pixi's output and build logs as Pixi runs, set `options(rpix.verbose = TRUE)`.

For more information, see <https://pixi.prefix.dev/latest/reference/cli/pixi/add/>.

Usage

```
pixi_add(
  packages,
  versions = NULL,
  channel = NULL,
  feature = NULL,
  platform = NULL,
  type = NULL,
  path = NULL,
  dry_run = FALSE
)
```

Arguments

<code>packages</code>	Package names. A version constraint can follow the name, as in " <code>dplyr>=1.1</code> ".
<code>versions</code>	Optional. Version constraints, either one for all packages or one per package (use <code>NA</code> for no constraint). A version without an operator, such as " <code>1.1</code> ", means <code>1.1.*</code> . For a package from GitHub, it's a branch, tag or commit instead, as with <code>@ref</code> .
<code>channel</code>	Optional. A conda channel to install the packages from. It's added to the project's channels if it isn't there yet.

feature	Optional. The feature to add the packages to, rather than the default one. See pixi_add_environment() .
platform	Optional. Only add the packages for this platform, such as "linux-64".
type	Optional. In a package, also add the packages to this field of DESCRIPTION, as <code>usethis::use_package()</code> does: "Imports", "Depends", "Suggests", "LinkingTo" or "Enhances" for R packages, or "SystemRequirements" for software outside R, given as conda packages ("conda::gdal"). A <code>>=</code> constraint becomes the minimum version there; other constraints only go in <code>pixi.toml</code> . A package from GitHub also goes in Remotes (unless it's CRAN's mirror, <code>cran/</code>). <code>SystemRequirements</code> gets conda-forge's names, such as <code>gdal</code> or <code>pkg-config</code> , added after what it says already, which stays as it is. DESCRIPTION changes once Pixi has added the packages, and feature still decides where they go in <code>pixi.toml</code> .
path	The project. Defaults to the project of the running Pixi environment, or the working directory.
dry_run	If TRUE, show the Pixi commands without running them.

Value

The commands (invisibly) if `dry_run = TRUE`, otherwise the result of the Pixi call (invisibly).

Examples

```
# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*" '
  ),
  file.path(project, "pixi.toml")
)

pixi_add("tibble", path = project, dry_run = TRUE)
pixi_add(
  c("dplyr>=1.1", "bioc::DESeq2", "conda::gdal"),
  path = project,
  dry_run = TRUE
)
pixi_add("dplyr", versions = "1.1", path = project, dry_run = TRUE)
```

pixi_add_channel	<i>Add or remove the project's channels and platforms</i>
------------------	---

Description

- `pixi_add_channel()` adds conda channels, which packages are installed from. `pixi_add()` adds the ones it needs itself.
- `pixi_add_platform()` adds platforms, such as "linux-64", "osx-arm64" or "win-64", so `pixi.lock` covers them too and collaborators on them get the same packages.
- `pixi_remove_channel()` and `pixi_remove_platform()` remove them again, e.g. a platform no one uses, which Pixi then no longer solves for, or a channel the project no longer needs packages from.

For more information, see <https://pixi.prefix.dev/latest/reference/cli/pixi/workspace/channel/> and <https://pixi.prefix.dev/latest/reference/cli/pixi/workspace/platform/>.

Usage

```
pixi_add_channel(channels, path = NULL, dry_run = FALSE)

pixi_add_platform(platforms, path = NULL, dry_run = FALSE)

pixi_remove_channel(channels, path = NULL, dry_run = FALSE)

pixi_remove_platform(platforms, path = NULL, dry_run = FALSE)
```

Arguments

<code>channels</code>	Channel names, such as "bioconda", or URLs.
<code>path</code>	The project. Defaults to the project of the running Pixi environment, or the working directory.
<code>dry_run</code>	If TRUE, show the Pixi command without running it.
<code>platforms</code>	Platform names.

Value

The command (invisibly) if `dry_run = TRUE`, otherwise the result of the Pixi call (invisibly).

Examples

```
# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
```

```

    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*" '
  ),
  file.path(project, "pixi.toml")
)

pixi_add_channel("bioconda", path = project, dry_run = TRUE)
pixi_add_platform("osx-64", path = project, dry_run = TRUE)

```

`pixi_auth_login` *Log in to private channels*

Description

Experimental. These may change, e.g. to pass the secret to Pixi on standard input once it can read it from there. See <https://github.com/roaldarbol/rpix/issues/97>.

Pixi needs credentials to install packages from private conda channels, e.g. on prefix.dev, anaconda.org, [Artifactory](https://artifactory.com) or [S3](https://s3.amazonaws.com). These functions run `pixi auth`, which stores credentials in your system's keychain, or in `~/.rattler/credentials.json`, for every Pixi project to use.

- `pixi_auth_login()` logs in to a host. It asks for the token or password with hidden input, so it isn't shown, or saved in your R history.
- `pixi_auth_logout()` removes the credentials for a host.
- `pixi_auth_status()` lists the hosts Pixi has credentials for, without the secrets.

`rpix` never shows or saves a secret itself. Pixi takes it as a command-line argument, so while Pixi runs, other users on the same computer could see it in the list of running processes.

For continuous integration, use the `auth-*` inputs of the `setup-pixi` action instead. For more information, see <https://pixi.prefix.dev/latest/deployment/authentication/>.

Usage

```

pixi_auth_login(
  host,
  method = c("token", "password", "conda-token", "s3"),
  username = NULL
)

pixi_auth_logout(host)

pixi_auth_status()

```

Arguments

host	The host, such as "prefix.dev" or "repo.example.com", or "s3://my-bucket" for a channel on S3.
method	How to log in: • "token": a token, e.g. for prefix.dev. • "password": a username and password (basic HTTP authentication), e.g. for Artifactory or Nexus. • "conda-token": a token for anaconda.org or quetz. • "s3": an access key ID and secret access key, for channels on S3.
username	For method = "password", the username. Asked for if not given.

Value

- `pixi_auth_login()` and `pixi_auth_logout()`: the result of the Pixi call, invisibly, with any secret hidden.
- `pixi_auth_status()`: a data frame with a row per host, and the columns host, method, username and source.

Examples

```
## Not run:
pixi_auth_login("prefix.dev")
pixi_auth_login("repo.example.com", method = "password", username = "me")
pixi_auth_status()
pixi_auth_logout("prefix.dev")

## End(Not run)
```

<code>pixi_check_matrix</code>	<i>Test a package in several environments</i>
--------------------------------	---

Description

For package developers: run the tests or R CMD check in several of the project's environments, e.g. one for each version of R, and see the results side by side.

- `pixi_check_matrix()` runs them. Each environment needs the package's dependencies, and testthat and pkgload for the tests, or rcmdcheck for R CMD check.
- `use_pixi_check_matrix()` adds an environment for each version of R, with those packages, named after the version: r44 for R 4.4. With `minimum = TRUE`, it also adds a minimum environment, with the oldest versions the package's DESCRIPTION allows.

Usage

```

pixi_check_matrix(
  environments = NULL,
  what = c("test", "check"),
  task = NULL,
  parallel = FALSE,
  path = NULL
)

use_pixi_check_matrix(r_versions = NULL, minimum = FALSE, path = NULL)

```

Arguments

<code>environments</code>	Optional. The environments. Defaults to all of them.
<code>what</code>	"test" to run the tests with <code>testthat::test_local()</code> , or "check" to run R CMD check with <code>rcmdcheck::rcmdcheck()</code> .
<code>task</code>	Optional. A Pixi task to run instead, such as "check". It passes if it succeeds.
<code>parallel</code>	If TRUE, run in all environments at once.
<code>path</code>	The project. Defaults to the project of the running Pixi environment, or the working directory.
<code>r_versions</code>	Versions of R, such as "4.4". Optional with <code>minimum = TRUE</code> .
<code>minimum</code>	If TRUE, also add a minimum environment; see below.

Value

- `pixi_check_matrix()`: a data frame with a row per environment, and the columns environment, r_version, ok, the counts (passed, failed, skipped and warnings for tests; errors, warnings and notes for R CMD check), seconds, and output: what it printed.
- `use_pixi_check_matrix()`: the names of the environments, invisibly.

The minimum environment

Checks that the minimum versions in DESCRIPTION really work: R at the version Depends asks for, and each package in Depends, Imports, LinkingTo and Suggests at the lowest version its constraint allows (Pixi's solve-strategy = "lowest-direct"; their own dependencies are the newest that fit).

- A package in Depends, Imports or LinkingTo without a minimum gets the oldest version on conda-forge for that R, which may be much older than you meant; rpix lists them.
- A Suggests without a minimum isn't what's tested, so it gets at least the version the project has, in its test environment or else the default one. rpix lists those the project doesn't have, which get the oldest version, as do the others if those versions don't solve for that R.
- `testthat`, `pkgload` and `rcmdcheck`, which run the checks, get recent minimums, or a higher one from DESCRIPTION.
- Without a minimum R in Depends, it uses the oldest of `r_versions`, or the project's R.

- Only versions on conda-forge (and bioconda) count, for the project's platforms. An old build's own dependencies are as conda-forge recorded them, which for old builds is sometimes looser than its DESCRIPTION: then R refuses to load it, although Pixi installed it.

Examples

```
## Not run:
use_pxi_check_matrix(c("4.3", "4.4", "4.5"))
use_pxi_check_matrix(minimum = TRUE)
pxi_check_matrix()
pxi_check_matrix(what = "check", parallel = TRUE)

## End(Not run)
```

pixi_environments *Features and environments*

Description

A Pixi project can have several environments, each with its own R and packages: say one with R 4.4 and one with R 4.5, or one with the packages your tests need on top of the project's own.

Environments are made of *features*: named sets of packages. The default feature holds what `pxi_add()` adds, and is in every environment unless it's left out. Add packages to another feature with `pxi_add(feature = )`; that also creates the feature. Then make an environment of it with `pxi_add_environment()`, and start its R with `pxi run --environment <name> R`.

- `pxi_environments()` lists the project's environments.
- `pxi_add_environment()` adds an environment, made of features.
- `pxi_remove_environment()` removes one.

For more information, see https://pixi.prefix.dev/latest/workspace/multi_environment/.

Usage

```
pxi_environments(path = NULL)

pxi_add_environment(
  name,
  features = NULL,
  r_version = NULL,
  solve_group = NULL,
  default_feature = TRUE,
  overwrite = FALSE,
  path = NULL,
  dry_run = FALSE
)

pxi_remove_environment(name, path = NULL, dry_run = FALSE)
```

Arguments

path	The project. Defaults to the project of the running Pixi environment, or the working directory.
name	The environment's name.
features	The features it's made of. They have to exist already, so add packages to them first with <code>pixi_add(feature =)</code> . Without any, the environment is a copy of the default one.
r_version	Optional. Another version of R for the environment, such as "4.4", in one step: a feature named after the environment gets that R, the default feature's packages (by name, so it gets the versions built for that R) and <code>conda-ecosystem-user-package-isolation</code> . The environment leaves out the default feature, which has the project's R, and is installed, with <code>rpix</code> in it if the project has <code>rpix</code> . Packages built from source aren't copied; <code>rpix</code> says how to add them. If that R can't have the project's packages, <code>pixi.toml</code> and <code>pixi.lock</code> are left as they were.
solve_group	Optional. Environments in the same solve group get the same versions of the packages they share.
default_feature	If FALSE, leave out the default feature. Needed when a feature conflicts with it, e.g. <code>pins</code> another version of R. If the environment has R, <code>rpix</code> then adds <code>conda-ecosystem-user-package-isolation</code> to its first feature, as <code>use_pixi()</code> does to the default one, to keep your personal R library out.
overwrite	If TRUE, replace an environment that already exists.
dry_run	If TRUE, show the Pixi command without running it.

Value

- `pixi_environments()`: a data frame with a row per environment, and the columns `name`, `features` and `platforms` (lists of character vectors) and `solve_group`.
- The others: the command (invisibly) if `dry_run = TRUE`, otherwise the result of the Pixi call (invisibly).

Examples

```
# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*"'
```

```

    ),
    file.path(project, "pixi.toml")
)

# An environment with R 4.4 and the project's packages, next to the
# default one
pixi_add_environment("r44", r_version = "4.4", path = project, dry_run = TRUE)

# An environment with the default packages plus testthat, on the same
# versions as the default environment
pixi_add("testthat", feature = "test", path = project, dry_run = TRUE)
pixi_add_environment(
  "test",
  features = "test",
  solve_group = "default",
  path = project,
  dry_run = TRUE
)

## Not run:
# Needs Pixi
pixi_environments()

## End(Not run)

```

`pixi_import_description`

Add a package's dependencies from its DESCRIPTION

Description

For package developers: add what the package in the project needs to the project, from its DESCRIPTION.

- Depends and Imports (and LinkingTo) go into the default feature, with their version constraints.
- Suggests go into a test feature, with a test environment of the default and test features. It's in the same solve group as the default environment, so it tests the same versions.
- Depends: R (≥ 4.1) becomes a constraint on R, if the project doesn't have R yet.

Each package is looked up on conda-forge, as `r-<name>`, and then on bioconda, as `bioconductor-<name>`. Packages that are on neither are left out, and listed.

Usage

```
pixi_import_description(test = TRUE, path = NULL, dry_run = FALSE)
```

Arguments

test	If TRUE, add the Suggests to a test feature and environment. If FALSE, leave them out.
path	The project. Defaults to the project of the running Pixi environment, or the working directory.
dry_run	If TRUE, show the Pixi command without running it.

Value

A list with the package specs added to the default feature (packages) and to the test feature (test), and the packages that weren't found (missing), invisibly.

Examples

```
## Not run:
pixi_import_description()

## End(Not run)
```

pixi_import_renv	<i>Move a project from renv</i>
------------------	---------------------------------

Description

Add the packages in a project's `renv.lock` to its Pixi project, and turn `renv` off.

- Packages from CRAN, or another CRAN-like repository, are looked up on conda-forge, and Bioconductor packages on bioconda. CRAN packages that aren't on conda-forge can be built from CRAN's source instead (experimental): `pixi_import_renv()` offers to, or if it can't ask, lists them with the command that does. Other packages that aren't on conda-forge or bioconda are left out, and listed.
- Packages from GitHub (experimental) are built from source at the commit `renv.lock` records, as `pixi_add("github:user/repo@commit")` does. With `versions = "none"`, they're built from the newest commit of the branch it records. Packages in a subfolder of a repository, and packages from elsewhere, are left out, and listed.
- By default only the packages that no other package in the lock file needs are added, so `pixi.toml` lists what the project uses, rather than everything those packages need too. Packages from GitHub are added either way, as conda-forge doesn't have them.
- The lock file's version of R becomes a constraint on R, if the project doesn't have R yet. If conda-forge doesn't have the packages for that R, e.g. a new R it hasn't built packages for yet, it says so and uses the newest R that has them.

It says which packages Pixi builds from source, as building them is experimental; see `pixi_add()`. `renv`'s `.Rprofile` line, `source("renv/activate.R")`, makes R use `renv`'s library, so `deactivate = TRUE` removes it, like `renv::deactivate()`. `renv.lock` and the `renv` folder stay; delete them when you no longer need them.

Usage

```
pixi_import_renv(  
  lockfile = NULL,  
  versions = c("minimum", "exact", "none"),  
  all = FALSE,  
  deactivate = TRUE,  
  path = NULL,  
  dry_run = FALSE  
)
```

Arguments

lockfile	Optional. The lock file. Defaults to the project's <code>renv.lock</code> .
versions	How to use the lock file's versions: • "minimum": at least that version, e.g. <code>>=1.1.4</code>. • "exact": that version, e.g. <code>==1.1.4</code>, if it's on conda-forge, and at least that version if it isn't. • "none": the newest versions.
all	If TRUE, add every package in the lock file, including those that are only there because other packages need them.
deactivate	If TRUE, remove <code>renv</code> 's line from the project's <code>.Rprofile</code> .
path	The project. Defaults to the project of the running Pixi environment, or the working directory.
dry_run	If TRUE, show the Pixi command without running it.

Value

A list, invisibly, with the package specs added (`packages`), the packages built from source (`from_source`), and the packages left out because `rpix` can't add them from where they came (`not_from_repository`) or they weren't found (`missing`).

Examples

```
## Not run:  
pixi_import_renv()  
pixi_import_renv(versions = "exact")  
  
## End(Not run)
```

pixi_info

Information about a Pixi project

Description

- `pixi_info()` returns what `pixi info` reports: the Pixi version and platform, the project, and its environments with their features, platforms, channels, tasks and location.
- `pixi_list()` lists the packages installed in an environment.
- `pixi_tree()` shows which packages depend on which.

For more information, see <https://pixi.prefix.dev/latest/reference/cli/pixi/info/>, <https://pixi.prefix.dev/latest/reference/cli/pixi/list/> and <https://pixi.prefix.dev/latest/reference/cli/pixi/tree/>.

Usage

```
pixi_info(path = NULL)
```

```
pixi_list(environment = NULL, explicit = FALSE, path = NULL)
```

```
pixi_tree(packages = NULL, environment = NULL, invert = FALSE, path = NULL)
```

Arguments

<code>path</code>	The project. Defaults to the project of the running Pixi environment, or the working directory.
<code>environment</code>	Optional. The environment. Defaults to default.
<code>explicit</code>	If TRUE, only list the packages in <code>pixi.toml</code> , not the ones they depend on.
<code>packages</code>	Optional. Only show these packages, and what they depend on. Names are translated like in <code>pixi_add()</code> .
<code>invert</code>	If TRUE, show what depends on packages instead.

Value

- `pixi_info()`: a list, as `pixi info --json` gives it.
- `pixi_list()`: a data frame with a row per package, and the columns `name`, `version`, `r_package` (the name of the R package, or NA), `explicit` (whether it's in `pixi.toml`), `channel` (its name, such as `conda-forge`), `kind` ("`conda`" or "`pypi`") and `build`.
- `pixi_tree()`: the tree's lines, invisibly.

Examples

```
## Not run:
pixi_info()$environments_info$name
pixi_list(explicit = TRUE)
pixi_tree("dplyr")

## End(Not run)
```

pixi_install

Install and update the project's packages

Description

- `pixi_install()` installs an environment as `pixi.lock` describes it, e.g. after cloning a project. `pixi run` does this by itself, too. If `pixi.lock` is out of date with `pixi.toml`, Pixi updates it first, unless locked or frozen says otherwise.
- `pixi_update()` updates packages to the newest versions `pixi.toml` allows, in `pixi.lock`, and installs them.
- `pixi_upgrade()` updates packages beyond that, and raises the versions in `pixi.toml` to match. It leaves R as it is, unless `r = TRUE`.
- `pixi_lock()` updates `pixi.lock` to match `pixi.toml`, without installing anything.

In an interactive session, `pixi_update()` and `pixi_upgrade()` first show what would change, and ask before going ahead. Then they install the environment this R runs in, or else the default one (or environment). If that's this R's own environment, restart R afterwards, as the packages it has loaded still use the old files: in RStudio, they offer to. On Windows, a running R keeps Pixi from replacing its packages' files, so there they only update `pixi.lock`, and say how to install it after closing R.

Upgrading R (`r = TRUE`) means every R package needs a build for the new R. conda-forge has fewer packages for a new version of R at first, and packages built from source are pinned to the R they were added for, so R only moves as far as all the project's packages allow.

Package names are translated like in `pixi_add()`. If Pixi skips an install script that a package needs, such as the ones Bioconductor's data packages download their data with, these functions offer to allow them in the project, as `pixi_add()` does.

For more information, see <https://pixi.prefix.dev/latest/reference/cli/pixi/install/>, <https://pixi.prefix.dev/latest/reference/cli/pixi/update/>, <https://pixi.prefix.dev/latest/reference/cli/pixi/upgrade/> and <https://pixi.prefix.dev/latest/reference/cli/pixi/lock/>.

Usage

```
pixi_install(
  environment = NULL,
  all = FALSE,
  locked = FALSE,
  frozen = FALSE,
```

```

    path = NULL,
    dry_run = FALSE
  )

  pixi_update(packages = NULL, environment = NULL, path = NULL, dry_run = FALSE)

  pixi_upgrade(
    packages = NULL,
    feature = NULL,
    r = FALSE,
    path = NULL,
    dry_run = FALSE
  )

  pixi_lock(path = NULL, dry_run = FALSE)

```

Arguments

<code>environment</code>	Optional. The environment. Defaults to <code>default</code> .
<code>all</code>	If <code>TRUE</code> , install all the project's environments.
<code>locked</code>	If <code>TRUE</code> , fail if <code>pixi.lock</code> is out of date with <code>pixi.toml</code> , rather than updating it, e.g. to check that a collaborator or CI gets exactly what's committed.
<code>frozen</code>	If <code>TRUE</code> , install exactly what <code>pixi.lock</code> says, without checking it against <code>pixi.toml</code> .
<code>path</code>	The project. Defaults to the project of the running Pixi environment, or the working directory.
<code>dry_run</code>	If <code>TRUE</code> , show the Pixi command without running it.
<code>packages</code>	Optional. Only update these packages. Defaults to all.
<code>feature</code>	Optional. Only upgrade the packages of this feature.
<code>r</code>	If <code>TRUE</code> , upgrade R too (<code>r-base</code>), as far as the project's packages allow. Naming " <code>r-base</code> " in packages does the same.

Value

The command (invisibly) if `dry_run = TRUE`, otherwise the result of the Pixi call (invisibly).

Examples

```

# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    ""
  ),
  project
)

```

```

    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*"'
  ),
  file.path(project, "pixi.toml")
)

pixi_install(path = project, dry_run = TRUE)
pixi_update("dplyr", path = project, dry_run = TRUE)
pixi_upgrade(path = project, dry_run = TRUE)

```

pixi_r

Run a function in another environment's R

Description

Call a function in the R of another Pixi environment, e.g. one with another version of R, and get its result back, like `callr::r()`. The function runs in a new R process, started with `pixi run`, so it uses that environment's R and packages; your current R is left alone.

The function and its arguments are copied to the new R, but not the variables around it. Refer to packages with `::` or load them in the function, and pass in what it needs as arguments.

Usage

```
pixi_r(func, args = list(), environment = NULL, show = TRUE, path = NULL)
```

Arguments

<code>func</code>	A function.
<code>args</code>	A list of arguments to call it with.
<code>environment</code>	Optional. The environment. Defaults to default.
<code>show</code>	If TRUE, show the function's output while it runs.
<code>path</code>	The project. Defaults to the project of the running Pixi environment, or the working directory.

Value

The function's result.

Examples

```

## Not run:
pixi_r(function() R.version.string, environment = "r44")
pixi_r(function(x) packageVersion(x), list("dplyr"), environment = "r44")

## End(Not run)

```

pixi_remove	<i>Remove packages</i>
-------------	------------------------

Description

Remove packages from the Pixi manifest. Package names are translated like in `pixi_add()`: "dplyr" removes r-dplyr and "bioc::DESeq2" removes bioconductor-deseq2. "DESeq2" removes bioconductor-deseq2 too, when that's what the manifest has, as `pixi_add()` adds it.

For more information, see <https://pixi.prefix.dev/latest/reference/cli/pixi/remove/>.

Usage

```
pixi_remove(
  packages,
  feature = NULL,
  platform = NULL,
  type = NULL,
  path = NULL,
  dry_run = FALSE
)
```

Arguments

packages	Package names.
feature	Optional. The feature to add the packages to, rather than the default one. See <code>pixi_add_environment()</code> .
platform	Optional. Only add the packages for this platform, such as "linux-64".
type	Optional. In a package, also remove the packages from this field of DESCRIPTION, such as "Imports" or "SystemRequirements", and packages from GitHub from Remotes.
path	The project. Defaults to the project of the running Pixi environment, or the working directory.
dry_run	If TRUE, show the Pixi command without running it.

Value

The command (invisibly) if `dry_run = TRUE`, otherwise the result of the Pixi call (invisibly).

Examples

```
# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
```

```

    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*"
  ),
  file.path(project, "pixi.toml")
)

pixi_remove("tibble", path = project, dry_run = TRUE)

```

pixi_scan

Find packages the project's code uses

Description

Look through the project's R code for the packages it uses, and compare them with the packages in `pixi.toml`:

- Packages the code uses, but that aren't in `pixi.toml`.
- R packages in the default environment that the code never uses.

It reads `.R` files, and the R chunks of `.qmd` and `.Rmd` files, and finds `library()`, `require()`, `requireNamespace()`, `loadNamespace()` and `pkg::fun`. It skips the `.pixi`, `renv`, `docs` and `.git` folders. It also finds `pkg::fun` in the commands of the project's Pixi tasks, such as `Rscript -e 'devtools::test()'`.

A package that only other packages need doesn't count as unused if it's in `pixi.toml` on purpose, so check before removing one. `rpix` and the packages it needs aren't listed as unused.

Usage

```
pixi_scan(add = FALSE, path = NULL)
```

Arguments

<code>add</code>	If <code>TRUE</code> , add the missing packages that are on <code>conda-forge</code> or <code>bioconda</code> .
<code>path</code>	The project. Defaults to the project of the running Pixi environment, or the working directory.

Value

A list, invisibly, with:

- `used`: a data frame with a row per package the code uses, and the columns `package` and `files` (a list of the files that use it).
- `missing`: the packages the code uses that aren't in `pixi.toml`.
- `unused`: the R packages in the default environment the code doesn't use, as conda names.

Examples

```
## Not run:
pixi_scan()
pixi_scan(add = TRUE)

## End(Not run)
```

pixi_search	<i>Search for packages</i>
-------------	----------------------------

Description

Show the available versions of a conda package and their dependencies.

The package name is translated like in `pixi_add()`: "dplyr" searches for r-dplyr, "bioc:DESeq2" for bioconductor-deseq2 on bioconda, and "conda:gdal" for gdal. A package that isn't on conda-forge is looked for on bioconda too, so "DESeq2" works as well. An R package's name has to match exactly, while Pixi also finds conda packages whose names contain the one searched for.

For more information, see <https://pixi.prefix.dev/latest/reference/cli/pixi/search/>.

Usage

```
pixi_search(package, channel = NULL, path = NULL, dry_run = FALSE)
```

Arguments

package	A package name.
channel	Optional. A conda channel to search. Inside a project, the project's channels are searched by default, and conda-forge otherwise.
path	The project. Defaults to the project of the running Pixi environment, or the working directory.
dry_run	If TRUE, show the Pixi command without running it.

Value

The command (invisibly) if `dry_run = TRUE`, otherwise the result of the Pixi call (invisibly).

Examples

```
# With `dry_run = TRUE`, rpix shows the Pixi command it would run
pixi_search("tibble", dry_run = TRUE)
pixi_search("bioc:DESeq2", dry_run = TRUE)
pixi_search("conda:numpy", channel = "conda-forge", dry_run = TRUE)
```

pixi_set_date	<i>Use packages as they were on a date</i>
---------------	--

Description

Make Pixi leave out every package published after a date, with the project's `exclude-newer` setting. The project then gets packages as they were on that day, like a snapshot of a package repository: useful to rerun an old analysis with the versions it was written for, or to check that a package still works with what users had a year ago. `pixi_set_date()` sets it, and updates `pixi.lock` to match.

A duration, such as `"7d"`, leaves out packages published in the last week instead, e.g. to stay clear of a release that turns out to be broken or compromised. `pixi_set_date(NULL)` removes the setting; packages stay as they are until you update them with `pixi_update()`.

Only packages that were on conda-forge (or the project's other channels) at the time are available, even if they were on CRAN then.

Usage

```
pixi_set_date(date, path = NULL)
```

Arguments

date	A date, such as <code>"2025-06-01"</code> or <code>Sys.Date() - 365</code> , a timestamp (<code>"2025-06-01T12:00:00Z"</code>), or a duration, such as <code>"7d"</code> . <code>NULL</code> removes the setting.
path	The project. Defaults to the project of the running Pixi environment, or the working directory.

Details

`pixi_add()` pins the versions it adds, so an existing project's pins may be newer than an earlier date. Pixi then can't solve the project, and `pixi.toml` is left as it was. Setting the date when you create the project, with `use_pixi(date = )`, avoids that.

The date applies to Pixi's build backend too, so packages built from source (`pixi_add("github:...")`) need a date of 2026-10-15 or later, when `pixi-build-r 0.2.0` was out, or a duration.

See https://pixi.prefix.dev/latest/reference/pixi_manifest/#exclude-newer-optional.

Value

The previous setting, or `NULL`, invisibly.

Examples

```
## Not run:
pixi_set_date("2025-06-01")
pixi_set_date("7d")
pixi_set_date(NULL)

## End(Not run)
```

Description

Check that R, its packages and the IDE use the project's Pixi environment, and suggest fixes for what doesn't. It reports:

- Pixi: where it is, and its version.
- The project, and whether `pixi.lock` is up to date with `pixi.toml`.
- Whether the project was moved or copied: Pixi's activation cache then has the old location, so it's removed, and environments installed in the old location need reinstalling.
- The date the project's packages are as of, if it has one (see `pixi_set_date()`).
- In a package, whether `pixi.toml` has what its DESCRIPTION needs: the Depends, Imports and LinkingTo packages in the default environment, the Suggests somewhere (e.g. a test environment), and installed versions at least the minimums DESCRIPTION asks for.
- In a package, whether `.Rbuildignore` leaves Pixi's files out of its builds.
- Environments with R that don't keep your personal R library out, e.g. ones that leave out the default feature.
- Private channels that refuse access, and the hosts Pixi has logins for (see `pixi_auth_login()`).
- Whether the running R is the R of the project's environment, and whether the environment is activated.
- Libraries and loaded packages from outside the project, such as your personal library. R's base packages come with every R, so aren't listed.
- Packages in the environment that Pixi didn't install, e.g. with `utils::install.packages()`, so `pixi.toml` doesn't record them.
- Whether the project's `.Rprofile` activates the environment when an IDE starts R directly.
- Whether the IDE you're in is set up for the project: it runs the project's R, or is set up by `use_pixi()`. If it's set up but runs another R, how to start the project's R in it.

Usage

```
pixi_sitrep(path = NULL)
```

Arguments

path	The project. Defaults to the project of the running Pixi environment, or the working directory.
------	---

Value

A list with what it found, invisibly.

Examples

```
# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*" '
  ),
  file.path(project, "pixi.toml")
)

pixi_sitrep(path = project)
```

pixi_switch

Switch to another environment's R

Description

Move your work to the R of another of the project's environments, e.g. one with another version of R. A running R can't change its own version or library, so this starts the other R the way your IDE needs:

- RStudio: starts a new RStudio with the environment's R, in the project. Close the old one when you're done with it.
- Positron: looks for interpreters again, and opens the interpreter picker, to pick the environment's R.
- VS Code, and editors based on it: points the R extension at the environment's R, in the project's .vscode/settings.json, keeping its console (R, arf or radian; see [use_pixi_vscode\(\)](#)). Reload the window to use it. On Windows, start the editor from the environment instead.
- Emacs, Neovim and Zed: points the project's .envrc at the environment (see [use_pixi_direnv\(\)](#)). Reload direnv in the editor, and restart R. On Windows, start the editor from the environment instead.
- Elsewhere: shows the command that starts the environment's R.

The environment is installed first, if it isn't yet.

Usage

```
pixi_switch(environment = NULL, ide = NULL, path = NULL)
```

Arguments

environment	The environment. In an interactive session, leave it out to pick one from a menu.
ide	Optional. The IDE: "rstudio", "positron", "vscode", "emacs", "neovim" or "zed". Defaults to the one you're in.
path	The project. Defaults to the project of the running Pixi environment, or the working directory.

Value

The environment's R, invisibly, or NULL if you cancel the menu.

Examples

```
## Not run:
pixi_switch("r44")
# Pick from a menu
pixi_switch()

## End(Not run)
```

`pixi_tasks`

Pixi tasks

Description

Tasks are commands saved in `pixi.toml`, such as `test = "Rscript -e 'devtools::test()'"`, which anyone with the project can run with `pixi run test`.

- `pixi_tasks()` lists the project's tasks.
- `pixi_run()` runs a task, showing its output as it runs.
- `pixi_add_task()` adds a task.
- `pixi_remove_task()` removes tasks.

For more information, see https://pixi.prefix.dev/latest/workspace/advanced_tasks/.

Usage

```
pixi_tasks(environment = NULL, path = NULL)
```

```
pixi_run(task, args = NULL, environment = NULL, path = NULL, dry_run = FALSE)
```

```
pixi_add_task(
  name,
  command,
  description = NULL,
  depends_on = NULL,
```

```

    args = NULL,
    env = NULL,
    cwd = NULL,
    feature = NULL,
    platform = NULL,
    path = NULL,
    dry_run = FALSE
  )

  pixi_remove_task(
    names,
    feature = NULL,
    platform = NULL,
    path = NULL,
    dry_run = FALSE
  )

```

Arguments

environment	Optional. The environment to list or run tasks in. Defaults to all environments for <code>pixi_tasks()</code> , and to the default environment for <code>pixi_run()</code> .
path	The project. Defaults to the project of the running Pixi environment, or the working directory.
task, name	The task's name.
args	For <code>pixi_run()</code> , values for the task's arguments, or extra arguments for its command. For <code>pixi_add_task()</code> , the names of the task's arguments, which the command uses as <code>{{ name }}</code> .
dry_run	If TRUE, show the Pixi command without running it.
command	The command, as you'd type it in a terminal.
description	Optional. What the task does, shown by <code>pixi task list</code> .
depends_on	Optional. Tasks to run before this one.
env	Optional. Environment variables to set, as a named character vector.
cwd	Optional. The folder to run the command in, relative to the project.
feature	Optional. The feature the task belongs to, rather than the default one. See pixi_add_environment() .
platform	Optional. Only add or remove the task for this platform, such as "win-64".
names	The tasks' names.

Value

- `pixi_tasks()`: a data frame with a row per task, and the columns `name`, `command`, `description`, `feature`, `environments`, `depends_on` and `args` (the last three are lists of character vectors). It includes the tasks for the platform you're on.
- The others: the command (invisibly) if `dry_run = TRUE`, otherwise the result of the Pixi call (invisibly).

Examples

```
# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*" '
  ),
  file.path(project, "pixi.toml")
)

pixi_add_task(
  "test",
  "Rscript -e 'devtools::test()'",
  path = project,
  dry_run = TRUE
)
pixi_run("test", path = project, dry_run = TRUE)
pixi_run("test", environment = "r44", path = project, dry_run = TRUE)

# A task with an argument
pixi_add_task(
  "render",
  "quarto render {{ file }}",
  args = "file",
  path = project,
  dry_run = TRUE
)
pixi_run("render", "analysis.qmd", path = project, dry_run = TRUE)

## Not run:
# Needs Pixi
pixi_tasks()

## End(Not run)
```

use_pixi

Set up a Pixi project for R

Description

Create a Pixi project in the working directory, or path, if there isn't one there, and add R and rpix to it.

It sets up that folder even inside another Pixi project. If a parent folder is a Pixi project, it says so, and in an interactive session asks whether to set that one up instead.

It also adds conda-forge's conda-ecosystem-user-package-isolation, which keeps your personal R library out of the environment's R. Otherwise conda-forge's R loads packages installed for your usual R first.

It can be run from any R. Afterwards, work in R started by Pixi: run `pixi run R` in a terminal, or use an IDE set up with `ide` (see <https://roald-arboel.com/rpix/articles/ide.html>). Each Pixi environment has its own R and package library, so `rpix` doesn't point a running R at a Pixi library: packages built for a different R can crash it.

It also adds a block to the project's `.Rprofile` that calls `pixi_activate()`, so the environment is activated when an IDE starts its R directly. Projects set up with `rpix` 0.3.0 or earlier have a "Pixi R library setup" block in their `.Rprofile` instead, which pointed a running R at the Pixi library. It's removed.

A new project supports Linux, macOS (Intel and Apple Silicon) and Windows, as well as this computer's platform, so collaborators on any of them can install it. `pixi.lock` then covers each of them, so a package can only be added if it's on conda-forge for all of them; for one that isn't, see platform in `pixi_add()`.

In a package, it adds `.pixi`, `pixi.toml` and `pixi.lock` to `.Rbuildignore`, so they stay out of the package's builds.

In a project that uses `renv` (with a `renv.lock`), it offers to move the packages over with `pixi_import_renv()` first, which also turns `renv` off. `renv` keeps `rpix` out of the project's own R, so run `use_pixi(path = )` from another R session, e.g. your usual R outside the project.

It turns on Pixi's activation cache for the project, in `.pixi/config.toml`. Pixi then reuses the environment's activation until the project changes, so R started through Pixi, e.g. by `pixi run` or Positron, starts in a fraction of a second rather than a few. The cache is experimental in Pixi, and doesn't notice changes outside the project, e.g. a newly installed Java.

Usage

```
use_pixi(
  r_version = NULL,
  ide = NULL,
  init_if_missing = TRUE,
  install_rpix = TRUE,
  date = NULL,
  path = NULL,
  activation_cache = TRUE,
  platforms = NULL,
  python = FALSE,
  ...
)
```

Arguments

<code>r_version</code>	Optional. The R version to add, such as "4.5". Defaults to the newest one that conda-forge has R packages for; it has a new R before its packages are built for it.
------------------------	---

ide	Optional. IDEs to set the project up for: any of "rstudio", "positron", "vscode", "emacs", "neovim" and "zed" (see use_pixi_ide), and "jupyter" (see use_pixi_jupyter() , experimental).
init_if_missing	If TRUE, create a Pixi project if there isn't one.
install_rpix	If TRUE, add rpix to the project (r-rpix), so it's in pixi.lock like the other packages, and a collaborator's environment has it too. Until rpix is on conda-forge, it comes from the rpix-forge channel, which is added to the project's channels after conda-forge, so everything else still comes from conda-forge. To try a change to rpix, set options(rpix.ref) to a branch, tag or commit, e.g. "main" or a pull request's branch: rpix is then also installed from GitHub, unless the environment has that commit already.
date	Optional. Use packages as they were on this date, such as "2025-06-01", e.g. to rerun an old analysis. Set before anything is added, so R and the packages are all as of that date. See pixi_set_date() .
path	The folder to set up. Defaults to the working directory.
activation_cache	If TRUE, turn on Pixi's activation cache for the project.
platforms	The platforms a new project supports, besides this computer's. Defaults to "linux-64", "osx-64", "osx-arm64" and "win-64". To add platforms to an existing project, use pixi_add_platform() .
python	If TRUE, add Python for reticulate, with use_pixi_python() (experimental).
...	Passed on to the IDE setup, e.g. console = "arf" for use_pixi_vscode() , or default_r = FALSE for use_pixi_positron() .

Value

The path to the project's manifest, invisibly.

Examples

```
## Not run:
use_pixi()
use_pixi(r_version = "4.5", ide = "positron")
use_pixi(ide = "vscode", console = "arf")
use_pixi(date = "2025-06-01")

## End(Not run)
```

Description

Write a `.envrc` file that activates the project's Pixi environment with **direnv** whenever you're in the project's folder: in a shell with `direnv`'s hook, and in editors that load `direnv`, such as Emacs, Neovim and Zed (see `use_pixi_emacs()`). R, languageserver and anything else started there then come from the environment.

If the project has a `.envrc` already, its other lines are kept. `direnv` only loads a `.envrc` you've allowed, and again after it changes: in an interactive session, `use_pixi_direnv()` offers to allow it. Install `direnv` with `pixi global install direnv`. In a package, `.envrc` is added to `.Rbuildignore`.

Usage

```
use_pixi_direnv(path = NULL, environment = "default")
```

Arguments

<code>path</code>	The project. Defaults to the working directory.
<code>environment</code>	The environment to activate.

Value

The `.envrc` file, invisibly.

Examples

```
# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*" '
  ),
  file.path(project, "pixi.toml")
)

use_pixi_direnv(path = project)
use_pixi_direnv(path = project, environment = "r44")
```

```
use_pixi_github_action
```

Set up GitHub Actions with the project's Pixi environments

Description

Write a GitHub Actions workflow that installs the project's environments exactly as `pixi.lock` records them, with `setup-pixi`, so CI uses the same R and packages as your computer. It replaces `setup-r`, `setup-r-dependencies` and `setup-pandoc` from `r-lib/actions`, whose other actions run as usual on top.

- "check": R CMD check, with `r-lib's check-r-package`, for each environment on each operating system. The environments need `rcmdcheck`, as `use_pixi_check_matrix()`'s do.
- "test-coverage": test coverage, reported to Codecov. The environment needs `covr` and `xml2`.
- "pkgdown": build the pkgdown site and publish it to GitHub Pages. The environment needs `pkgdown` and `pandoc`.
- "task": run one of the project's Pixi tasks, e.g. to render a report.

In a package, "check", "test-coverage" and "pkgdown" also need the package's dependencies from DESCRIPTION, Suggests included. `rpix` warns about each environment that lacks any of these.

The operating systems are those of the project's platforms that GitHub runs: Linux (`linux-64`), macOS (`osx-arm64`) and Windows (`win-64`). Add platforms with `pixi_add_platform()`. Pixi is pinned to the version you have, and the workflow fails if `pixi.lock` is out of date with `pixi.toml`, so commit both.

Usage

```
use_pixi_github_action(
  workflow = c("check", "test-coverage", "pkgdown", "task"),
  environments = NULL,
  task = NULL,
  overwrite = FALSE,
  path = NULL
)
```

Arguments

<code>workflow</code>	Which workflow: "check", "test-coverage", "pkgdown" or "task".
<code>environments</code>	Optional. The environments to use; "test-coverage" and "pkgdown" use the first. Defaults to those that have what the workflow needs: for "check" all of them, and for "test-coverage" and "pkgdown" the first, or else the default environment. "task" defaults to the default environment.
<code>task</code>	For "task", the task to run.
<code>overwrite</code>	If TRUE, replace a workflow that's there.
<code>path</code>	The project. Defaults to the working directory.

Value

The path to the workflow file, invisibly.

Examples

```
## Not run:
use_pixi_github_action("check")
use_pixi_github_action("check", environments = c("r44", "r45"))
use_pixi_github_action("task", task = "render")

## End(Not run)
```

use_pixi_ide

Set up an IDE for a Pixi project

Description

Set an IDE or editor up to use the R in the project's Pixi environment. `use_pixi()` calls these with its `ide` argument.

- `use_pixi_rstudio()` adds, or replaces, an `rstudio` task in `pixi.toml` that starts a new RStudio with the environment's R and the project's `.Rproj` file, on Linux, macOS and Windows (`linux-64`, `osx-64`, `osx-arm64` and `win-64`). Pixi adds any of those platforms the project doesn't have. It creates the `.Rproj` file if there isn't one. Start RStudio with `pixi run rstudio`.
- `use_pixi_positron()` turns on Positron's discovery of R in Pixi environments, and makes the default environment's R Positron's default for the project, in `.vscode/settings.json`. Positron (2026.10 or later) then starts that R when the project opens, until you pick another R there. The path is for the operating system you run it on; on another one, Positron ignores it.
- `use_pixi_vscode()` adds `languageserver`, the R extension's `sess` (how it talks to R) and `jgd` (its plots), and the console if it isn't R, to the environment. `sess` and `jgd` come from the **pixi-forge** channel until they're on `conda-forge`; it's added to the project's channels after `conda-forge`. It points the R extension for VS Code, and the editors based on it such as VSCodium, Cursor and Windsurf, at the environment's R and the console, in `.vscode/settings.json`. On Windows, the environment's R only starts when the environment is activated, so it leaves R and the console to be found on the `PATH` instead: start VS Code with `pixi run code .` (or `codium`, `cursor` or `windsurf` instead of `code`).
- `use_pixi_emacs()`, `use_pixi_neovim()` and `use_pixi_zed()` add `languageserver` to the environment, and a `.envrc` that activates the environment with `direnv` (see `use_pixi_direnv()`). The editor, or the shell it starts from, then finds the environment's R and `languageserver` on the `PATH`: Emacs with its `envrc` package, Neovim from a shell with `direnv`'s hook or with a `direnv` plugin, and Zed by itself. On Windows, start the editor from the environment instead, e.g. with `pixi run nvim`.

In an interactive session, `use_pixi_vscode()` asks which console to use, unless `console` is given. Otherwise it keeps the console already set up, or R.

Existing settings in `.vscode/settings.json` are kept. In a package, `.vscode` is added to `.Rbuildignore`.

Usage

```

use_pixi_rstudio(path = NULL)

use_pixi_positron(path = NULL, default_r = TRUE)

use_pixi_vscode(path = NULL, console = NULL)

use_pixi_emacs(path = NULL)

use_pixi_neovim(path = NULL)

use_pixi_zed(path = NULL)

```

Arguments

path	The project. Defaults to the working directory.
default_r	For use_pixi_positron(): if TRUE, make the default environment's R Positron's default for the project.
console	For use_pixi_vscode(): the R console, "R", "arf" or "radian". NULL asks.

Value

The file they changed, invisibly.

Examples

```

# A throwaway Pixi project in R's temporary folder, so the examples leave
# yours alone. With `dry_run = TRUE`, rpix shows the Pixi command it would run.
project <- file.path(tempdir(), "rpix-example")
dir.create(project, showWarnings = FALSE)
writeLines(
  c(
    "[workspace]",
    'name = "rpix-example"',
    'channels = ["conda-forge"]',
    'platforms = ["linux-64", "osx-arm64", "win-64"]',
    "",
    "[dependencies]",
    'r-base = "4.5.*"',
    'conda-ecosystem-user-package-isolation = "*" '
  ),
  file.path(project, "pixi.toml")
)

use_pixi_positron(path = project)

## Not run:
# These need Pixi, and add packages to the project
use_pixi_rstudio()
use_pixi_vscode(console = "arf")

```

```
use_pxi_neovim()

## End(Not run)
```

use_pxi_jupyter

Use Jupyter notebooks with the project's R

Description

Experimental. This may change, see <https://github.com/roaldarbol/rpix/issues/184>.

Set the project up for Jupyter notebooks, so they run with the R of the project's environments:

- It adds a jupyter environment with JupyterLab, of its own, so the environments with R don't get Python and JupyterLab.
- It adds IRkernel (r-irkernel) to each environment with R, and, where an environment has Python, ipykernel.
- It writes a kernel for each of those environments into the project, in .jupyter/kernels/, named e.g. "R (Pixi: default)" or "R (Pixi: r44)". A kernel starts its environment's R with `pixi run --environment`, so Pixi activates the environment, also on Windows. The kernels have no paths of this computer in them, so commit them with the project.
- It adds a jupyter task that starts JupyterLab with those kernels: `pixi_run("jupyter")`, or `pixi run jupyter` in a terminal.

rpix doesn't use the `ir` kernel that conda-forge's IRkernel brings, as a kernel of that name in your own Jupyter folder, e.g. from `IRkernel::installspec()`, would be used instead.

Run it again after adding an environment, to add that environment's kernel.

Usage

```
use_pxi_jupyter(path = NULL)
```

Arguments

path	The project. Defaults to the project of the running Pixi environment, or the working directory.
------	---

Value

The path to the project's manifest, invisibly.

Examples

```
## Not run:
# Needs Pixi, and adds JupyterLab and IRkernel to the project
use_pxi_jupyter()
pixi_run("jupyter")

## End(Not run)
```

use_pixi_python

Use Python alongside R, for reticulate

Description

Experimental. This may change, see <https://github.com/roaldarbol/rpix/issues/108>.

Add Python to the project's environment, and point reticulate at it, so Python and its packages are in the same environment and `pixi.lock` as R and its packages.

- It adds python (at `python_version`, if given), `r-reticulate`, and for Python 3.15 or later `libpython`: conda-forge has Python's shared library, which reticulate loads, apart from Python since 3.15.
- It sets `RETICULATE_PYTHON` to the environment's Python, in `pixi.toml`'s `[activation.env]`, so R started with `pixi run` uses it, and so does R started by an IDE, through the project's `.Rprofile` (`pixi_activate()`). Without it, reticulate uses a Python of its own.
- It adds `micromamba`, and points `RETICULATE_CONDA` to it: reticulate activates a conda environment again with a conda program before it uses its Python, and a Pixi project doesn't have one (<https://github.com/rstudio/reticulate/issues/1650>).

Add Python packages with `pixi_add()`: `pixi_add("conda::numpy")` from conda-forge, or `pixi_add("pypi::pydytuesday")` from PyPI. `reticulate`'s `py_require()` doesn't install into the project's Python.

Restart R afterwards, so reticulate finds the environment's Python. The Windows settings are written only if the project has `win-64`: if you add it later, run `use_pixi_python()` again.

Usage

```
use_pixi_python(python_version = NULL, path = NULL)
```

Arguments

`python_version` Optional. The Python version, such as "3.13". Defaults to the newest one that solves with the project.

`path` The project. Defaults to the project of the running Pixi environment, or the working directory.

Value

The path to the project's manifest, invisibly.

Examples

```
## Not run:
# Needs Pixi, and adds Python to the project
use_pixi_python()
pixi_add(c("conda::numpy", "pypi::pydytuesday"))

## End(Not run)
```

Index

install_pixi, [2](#)

pixi_activate, [3](#)
pixi_activate(), [29](#), [36](#)
pixi_add, [4](#)
pixi_add(), [3](#), [7](#), [11](#), [14](#), [16](#), [17](#), [20](#), [22](#), [29](#), [36](#)
pixi_add_channel, [7](#)
pixi_add_environment
 (pixi_environments), [11](#)
pixi_add_environment(), [6](#), [20](#), [27](#)
pixi_add_platform (pixi_add_channel), [7](#)
pixi_add_platform(), [30](#), [32](#)
pixi_add_task (pixi_tasks), [26](#)
pixi_auth_login, [8](#)
pixi_auth_login(), [24](#)
pixi_auth_logout (pixi_auth_login), [8](#)
pixi_auth_status (pixi_auth_login), [8](#)
pixi_check_matrix, [9](#)
pixi_environments, [11](#)
pixi_import_description, [13](#)
pixi_import_description(), [3](#)
pixi_import_renv, [14](#)
pixi_import_renv(), [29](#)
pixi_info, [16](#)
pixi_install, [17](#)
pixi_list (pixi_info), [16](#)
pixi_lock (pixi_install), [17](#)
pixi_r, [19](#)
pixi_remove, [20](#)
pixi_remove(), [3](#), [5](#)
pixi_remove_channel (pixi_add_channel),
 [7](#)
pixi_remove_environment
 (pixi_environments), [11](#)
pixi_remove_platform
 (pixi_add_channel), [7](#)
pixi_remove_task (pixi_tasks), [26](#)
pixi_run (pixi_tasks), [26](#)
pixi_scan, [21](#)
pixi_search, [22](#)
pixi_set_date, [23](#)
pixi_set_date(), [5](#), [24](#), [30](#)
pixi_sitrep, [24](#)
pixi_switch, [25](#)
pixi_tasks, [26](#)
pixi_tree (pixi_info), [16](#)
pixi_update (pixi_install), [17](#)
pixi_update(), [3](#), [23](#)
pixi_upgrade (pixi_install), [17](#)
use_pixi, [28](#)
use_pixi(), [3](#), [5](#), [12](#), [24](#), [33](#)
use_pixi_check_matrix
 (pixi_check_matrix), [9](#)
use_pixi_check_matrix(), [32](#)
use_pixi_direnv, [30](#)
use_pixi_direnv(), [25](#), [33](#)
use_pixi_emacs (use_pixi_id), [33](#)
use_pixi_emacs(), [31](#)
use_pixi_github_action, [32](#)
use_pixi_id, [30](#), [33](#)
use_pixi_jupyter, [35](#)
use_pixi_jupyter(), [30](#)
use_pixi_neovim (use_pixi_id), [33](#)
use_pixi_positron (use_pixi_id), [33](#)
use_pixi_positron(), [30](#)
use_pixi_python, [36](#)
use_pixi_python(), [5](#), [30](#)
use_pixi_rstudio (use_pixi_id), [33](#)
use_pixi_vscode (use_pixi_id), [33](#)
use_pixi_vscode(), [25](#), [30](#)
use_pixi_zed (use_pixi_id), [33](#)