# Package 'pkgcache'

July 23, 2025

**Title** Cache 'CRAN'-Like Metadata and R Packages

**Version** 2.2.4

**Description** Metadata and package cache for CRAN-like repositories. This is a utility package to be used by package management tools that want to take advantage of caching.

**License** MIT + file LICENSE

**URL** https://r-lib.github.io/pkgcache/, https://github.com/r-lib/pkgcache

**BugReports** https://github.com/r-lib/pkgcache/issues

**Depends** R (>= 3.4)

**Imports** callr (>= 2.0.4.9000), cli (>= 3.2.0), curl (>= 3.2), filelock, jsonlite, processx (>= 3.3.0.9001), R6, tools, utils

**Suggests** covr, debugme, desc, fs, keyring, pillar, pingr, rprojroot, sessioninfo, spelling, testthat (>= 3.2.0), webfakes (>= 1.1.5), withr, zip

**Config/Needs/website** tidyverse/tidytemplate

**Config/testthat/edition** 3

**Config/usethis/last-upkeep** 2025-04-30

**Encoding** UTF-8

**Language** en-US

**RoxygenNote** 7.3.2.9000

**NeedsCompilation** yes

**Author** Gábor Csárdi [aut, cre], Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

**Maintainer** Gábor Csárdi <csardi.gabor@gmail.com>

**Repository** CRAN

**Date/Publication** 2025-05-26 14:10:02 UTC

# Contents



---

`pkgcache-package` *Cache for package data and metadata*

---

## Description

Metadata and package cache for CRAN-like repositories. This is a utility package to be used by package management tools that want to take advantage of caching.

## Details

Metadata and package cache for CRAN-like repositories. This is a utility package to be used by package management tools that want to take advantage of caching.

### Installation:

You can install the released version of pkgcache from CRAN with:

```
install.packages("pkgcache")
```

If you need the development version, you can install it from GitHub with:

```
pak::pak("r-lib/pkgcache")
```

**Metadata cache:**

`meta_cache_list()` lists all packages in the metadata cache. It includes Bioconductor package, and all versions (i.e. both binary and source) of the packages for the current platform and R version.

(We load the pillar package, because it makes the pkgcache data frames print nicer, similarly to tibbles.)

```
library(pkgcache)
library(pillar)
meta_cache_list()
#> # A data frame: 50,641 x 33
#>   package   version depends suggests license md5sum sha256sum needscompilation
#>     <chr>      <chr>   <chr>   <chr>    <chr>   <chr>  <chr>     <chr>
#>  1 A3         1.0.0   R (>= ~ randomF~ GPL (>~ 929a4~ "\n     ~ no
#>  2 AATtools   0.0.3   R (>= ~ <NA>     GPL-3   de2ec~ "\n     ~ no
#>  3 ABACUS     1.0.0   R (>= ~ rmarkdo~ GPL-3   28795~ "\n     ~ no
#>  4 ABC.RAP    0.9.0   R (>= ~ knitr, ~ GPL-3   0158e~ "\n     ~ no
#>  5 ABCanalys~ 1.2.1   R (>= ~ <NA>     GPL-3   4cbe1~ "\n     ~ no
#>  6 ABCoptim   0.15.0  <NA>    testtha~ MIT + ~ a294d~ "\n     ~ yes
#>  7 ABCp2      1.2     MASS    <NA>     GPL-2   d049b~  <NA>     no
#>  8 ABHgenoty~ 1.0.1   <NA>    knitr, ~ GPL-3   fce25~ "\n     ~ no
#>  9 ABM        0.4.3   <NA>    <NA>     GPL (>~ 7aaae~ "\n     ~ yes
#> 10 ABPS       0.3     <NA>    testthat GPL (>~ d3f00~ "\n     ~ no
#> # i 50,631 more rows
#> # i 25 more variables: imports <chr>, linkingto <chr>, archs <chr>,
#> #   enhances <chr>, license_restricts_use <chr>, priority <chr>, os_type <chr>,
#> #   license_is_foss <chr>, repodir <chr>, rversion <chr>, platform <chr>,
#> #   ref <chr>, type <chr>, direct <lgl>, status <chr>, target <chr>,
#> #   mirror <chr>, sources <list>, filesize <int>, sha256 <chr>, sysreqs <chr>,
#> #   built <chr>, published <dttm>, deps <list>, path <chr>
```

`meta_cache_deps()` and `meta_cache_revdeps()` can be used to look up dependencies and reverse dependencies.

The metadata is updated automatically if it is older than seven days, and it can also be updated manually with `meta_cache_update()`.

See the `cranlike_metadata_cache` R6 class for a lower level API, and more control.

**Package cache:**

Package management tools may use the `pkg_cache_*` functions and in particular the `package_cache` class, to make use of local caching of package files.

The `pkg_cache_*` API is high level, and uses a user level cache:

```
pkg_cache_summary()
#> $cachepath
#> [1] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/pkg"
#>
#> $files
#> [1] 475
#>
```

```
#> $size
#> [1] 669462030
```

```
pkg_cache_list()
#> # A data frame: 475 x 11
#>   fullpath   path  package url  etag  sha256 version platform built vignettes
#>   <chr>      <chr> <chr>   <chr> <chr> <chr>  <chr>   <chr>    <int>    <int>
#>  1 /Users/gab~ src/~ remotes <NA>  <NA>  5b7eb~ <NA>    <NA>         0       NA
#>  2 /Users/gab~ src/~ remotes <NA>  <NA>  5b7eb~ 2.5.0.~ source       1        0
#>  3 /Users/gab~ src/~ remotes <NA>  <NA>  5b7eb~ 2.5.0.~ aarch64~     1        0
#>  4 /Users/gab~ src/~ cli     <NA>  <NA>  9d6e9~ <NA>    <NA>         0       NA
#>  5 /Users/gab~ src/~ cli     <NA>  <NA>  9d6e9~ 3.6.3.~ source       1        0
#>  6 /Users/gab~ src/~ cli     <NA>  <NA>  9d6e9~ 3.6.3.~ aarch64~     1        0
#>  7 /Users/gab~ src/~ cli     <NA>  <NA>  8a0b8~ <NA>    <NA>         0       NA
#>  8 /Users/gab~ src/~ cli     <NA>  <NA>  8a0b8~ 3.6.3.~ source       1        0
#>  9 /Users/gab~ src/~ cli     <NA>  <NA>  8a0b8~ 3.6.3.~ aarch64~     1        0
#> 10 /Users/gab~ src/~ cli     <NA>  <NA>  2dd91~ <NA>    <NA>         0       NA
#> # i 465 more rows
#> # i 1 more variable: rversion <chr>
```

```
pkg_cache_find(package = "dplyr")
#> # A data frame: 2 x 11
#>   fullpath    path  package url  etag  sha256 version platform built vignettes
#>   <chr>       <chr> <chr>   <chr> <chr> <chr>  <chr>   <chr>    <int>    <int>
#> 1 /Users/gabo~ bin/~ dplyr   http~ "\"6~ 721c2~ 1.1.4   aarch64~   NA       NA
#> 2 /Users/gabo~ bin/~ dplyr   http~ "\"1~ 721c2~ 1.1.4   aarch64~   NA       NA
#> # i 1 more variable: rversion <chr>
```

`pkg_cache_add_file()` can be used to add a file, `pkg_cache_delete_files()` to remove files, `pkg_cache_get_files()` to copy files out of the cache.

The `package_cache` class provides a finer API.

**Installed packages:**

pkgcache contains a very fast DCF parser to parse `PACKAGES*` files, or the `DESCRIPTION` files in installed packages. `parse_packages()` parses all fields from `PACKAGES`, `PACKAGES.gz` or `PACKAGES.rds` files. `parse_installed()` reads *all* metadata from packages installed into a library:

```
parse_installed()
#> # A data frame: 606 x 125
#>   Package     Title   Version `Authors@R` Description URL   BugReports License
#>   <chr>       <chr>   <chr>   <chr>       <chr>       <chr> <chr>      <chr>
#>  1 AzureAuth   Authen~ 1.3.3   "c(\n   p~ "Provides ~ "htt~ https://g~ MIT + ~
#>  2 AzureGraph  Simple~ 1.3.4   "c(\n   p~ "A simple ~ "htt~ https://g~ MIT + ~
#>  3 AzureRMR    Interf~ 2.4.4   "c(\n   p~ "A lightwe~ "htt~ https://g~ MIT + ~
#>  4 AzureStor   Storag~ 3.7.0   "c(\n   p~ "Manage st~ "htt~ https://g~ MIT + ~
#>  5 BH          Boost ~ 1.84.0~ <NA>      "Boost pro~ "htt~ https://g~ BSL-1.0
#>  6 Biobase     Biobas~ 2.64.0  "c(\n   p~ "Functions~ "htt~ https://g~ Artist~
#>  7 BiocGenerics S4 gen~ 0.52.0  <NA>      "The packa~ "htt~ https://g~ Artist~
#>  8 BiocManager Access~ 1.30.23 "c(\n   p~ "A conveni~ "htt~ https://g~ Artist~
```

```
#>  9 BiocVersion  Set th~ 3.19.1  "c(\n    p~ "This pack~  <NA> <NA>      Artist~
#> 10 CompQuadForm Distri~ 1.4.3    <NA>      "Computes ~  <NA> <NA>      GPL (>~
#> # i 596 more rows
#> # i 117 more variables: VignetteBuilder <chr>, Depends <chr>, Imports <chr>,
#> #   Suggests <chr>, RoxygenNote <chr>, NeedsCompilation <chr>, Packaged <chr>,
#> #   Author <chr>, Maintainer <chr>, Repository <chr>, `Date/Publication` <chr>,
#> #   Built <chr>, RemoteType <chr>, RemotePkgRef <chr>, RemoteRef <chr>,
#> #   RemoteRepos <chr>, RemotePkgPlatform <chr>, RemoteSha <chr>, Type <chr>,
#> #   Date <chr>, biocViews <chr>, LazyLoad <chr>, Collate <chr>, ...
```

**Bioconductor support:**

Both the metadata cache and the package cache support Bioconductor by default, automatically. See the `BioC_mirror` option and the `R_BIOC_MIRROR` and `R_BIOC_VERSION` environment variables below to configure Bioconductor support.

**Package Options:**

- The `BioC_mirror` option can be used to select a Bioconductor mirror. This takes priority over the `R_BIOC_MIRROR` environment variable.
- You can use the `pkg.current_platform` option to set the platform string for the current platform for the `current_r_platform()` function. This is useful if pkgcache didn't detect the platform correctly. Alternatively, you can use the `PKG_CURRENT_PLATFORM` environment variable. The option takes priority.
- `pkgcache_timeout` is the HTTP timeout for all downloads. It is in seconds, and the limit for downloading the whole file. Defaults to 3600, one hour. It corresponds to the `TIMEOUT` libcurl option.
- `pkgcache_connecttimeout` is the HTTP timeout for the connection phase. It is in seconds and defaults to 30 seconds. It corresponds to the `CONNECTTIMEOUT` libcurl option.
- `pkgcache_low_speed_limit` and `pkgcache_low_speed_time` are used for a more sensible HTTP timeout. If the download speed is less than `pkgcache_low_speed_limit` bytes per second for at least `pkgcache_low_speed_time` seconds, the download errors. They correspond to the `LOW_SPEED_LIMIT` and `LOW_SPEED_TIME` curl options.

**Package environment variables:**

- The `R_BIOC_VERSION` environment variable can be used to override the default Bioconductor version detection and force a given version. E.g. this can be used to force the development version of Bioconductor.
- The `R_BIOC_MIRROR` environment variable can be used to select a Bioconductor mirror. The `BioC_mirror` option takes priority over this, if set.
- You can use the `PKG_CURRENT_PLATFORM` environment variable to set the platform string for the current platform for the `current_r_platform()` function. This is useful if pkgcache didn't detect the platform correctly. Alternatively, you can use the `pkg.current_platofrm` option, which takes. priority over the environment variable.
- `PKGCACHE_PPM_REPO` is the name of the Posit Package Manager repository to use. Defaults to `"cran"`.
- `PKGCACHE_PPM_URL` is the base URL of the Posit Package Manager instance to use. It defaults to the URL of the Posit Public Package Manager instance at `https://packagemanager.posit.co/client/#/`.

- PKGCACHE_TIMEOUT is the HTTP timeout for all downloads. It is in seconds, and the limit for downloading the whole file. Defaults to 3600, one hour. It corresponds to the TIMEOUT libcurl option. The pkgcache_timeout option has priority over this, if set.
- PKGCACHE_CONNECTTIMEOUT is the HTTP timeout for the connection phase. It is in seconds and defaults to 30 seconds. It corresponds to the CONNECTTIMEOUT libcurl option. The pkgcache_connecttimeout option takes precedence over this, if set.
- PKGCACHE_LOW_SPEED_LIMIT and PKGCACHE_LOW_SPEED_TIME are used for a more sensible HTTP timeout. If the download speed is less than PKGCACHE_LOW_SPEED_LIMIT bytes per second for at least PKGCACHE_LOW_SPEED_TIME seconds, the download errors. They correspond to the LOW_SPEED_LIMIT and LOW_SPEED_TIME curl options. The pkgcache_low_speed_time and pkgcache_low_speed_limit options have priority over these environment variables, if they are set.
- R_PKG_CACHE_DIR is used for the cache directory, if set. (Otherwise tools::R_user_dir("pkgcache", "cache") is used, see also meta_cache_summary() and pkg_cache_summary()).

**Using pkgcache in CRAN packages:**

If you use pkgcache in your CRAN package, please make sure that

- you don't use pkgcache in your examples, and
- you set the R_USER_CACHE_DIR environment variable to a temporary directory (e.g. via tempfile()) during test cases. See the tests/testthat/setup.R file in pkgcache for an example.

This is to make sure that pkgcache does not modify the user's files while running R CMD check.

**Code of Conduct:**

Please note that the pkgcache project is released with a Contributor Code of Conduct. By contributing to this project, you agree to abide by its terms.

**License:**

MIT (c) Posit Software, PBC

## Author(s)

**Maintainer**: Gábor Csárdi <csardi.gabor@gmail.com>

Other contributors:

- Posit Software, PBC (ROR) [copyright holder, funder]

## See Also

Useful links:

- https://r-lib.github.io/pkgcache/
- https://github.com/r-lib/pkgcache
- Report bugs at https://github.com/r-lib/pkgcache/issues

bioc_version *Query Bioconductor version information*

## Description

Various helper functions to deal with Bioconductor repositories. See https://www.bioconductor.org/ for more information on Bioconductor.

## Usage

```
bioc_version(r_version = getRversion(), forget = FALSE)

bioc_version_map(forget = FALSE)

bioc_devel_version(forget = FALSE)

bioc_release_version(forget = FALSE)

bioc_repos(bioc_version = "auto", forget = FALSE)
```

## Arguments

| | |
|---|---|
| r_version | The R version number to match. |
| forget | Use TRUE to avoid caching the Bioconductor mapping. |
| bioc_version | Bioconductor version string or package_version object, or the string "auto" to use the one matching the current R version. |

## Details

bioc_version() queries the matching Bioconductor version for an R version, defaulting to the current R version

bioc_version_map() returns the current mapping between R versions and Bioconductor versions.

bioc_devel_version() returns the version number of the current Bioconductor devel version.

bioc_release_version() returns the version number of the current Bioconductor release.

bioc_repos() returns the Bioconductor repository URLs.

See the BioC_mirror option and the R_BIOC_MIRROR and R_BIOC_VERSION environment variables in the pkgcache manual page. They can be used to customize the desired Bioconductor version.

## Value

bioc_version() returns a package_version object.

bioc_version_map() returns a data frame with columns:

- bioc_version: package_version object, Bioconductor versions.
- r_version: package_version object, the matching R versions.

- `bioc_status`: factor, with levels: `out-of-date`, `release`, `devel`, `future`.

`bioc_devel_version()` returns a package_version object.

`bioc_release_version()` returns a package_version object.

`bioc_repos()` returns a named character vector.

## Examples

```
bioc_version()
bioc_version("4.0")
bioc_version("4.1")


bioc_version_map()


bioc_devel_version()


bioc_release_version()


bioc_repos()
```

---

`cranlike_metadata_cache`

*Metadata cache for a CRAN-like repository*

---

## Description

This is an R6 class that implements the metadata cache of a CRAN-like repository. For a higher level interface, see the `meta_cache_list()`, `meta_cache_deps()`, `meta_cache_revdeps()` and `meta_cache_update()` functions.

## Details

The cache has several layers:

- The data is stored inside the `cranlike_metadata_cache` object.
- It is also stored as an RDS file, in the session temporary directory. This ensures that the same data is used for all queries of a `cranlike_metadata_cache` object.
- It is stored in an RDS file in the user's cache directory.
- The downloaded raw `PACKAGES*` files are cached, together with HTTP ETags, to minimize downloads.

It has a synchronous and an asynchronous API.

## Usage

```
cmc <- cranlike_metadata_cache$new(
  primary_path = NULL, replica_path = tempfile(),
  platforms = default_platforms(), r_version = getRversion(),
  bioc = TRUE, cran_mirror = default_cran_mirror(),
  repos = getOption("repos"),
  update_after = as.difftime(7, units = "days"))

cmc$list(packages = NULL)
cmc$async_list(packages = NULL)

cmc$deps(packages, dependencies = NA, recursive = TRUE)
cmc$async_deps(packages, dependencies = NA, recursive = TRUE)

cmc$revdeps(packages, dependencies = NA, recursive = TRUE)
cmc$async_revdeps(packages, dependencies = NA, recursive = TRUE)

cmc$update()
cmc$async_update()
cmc$check_update()
cmc$asnyc_check_update()

cmc$summary()

cmc$cleanup(force = FALSE)
```

## Arguments

- `primary_path`: Path of the primary, user level cache. Defaults to the user level cache directory of the machine.
- `replica_path`: Path of the replica. Defaults to a temporary directory within the session temporary directory.
- `platforms`: see `default_platforms()` for possible values.
- `r_version`: R version to create the cache for.
- `bioc`: Whether to include BioConductor packages.
- `cran_mirror`: CRAN mirror to use, this takes precedence over `repos`.
- `repos`: Repositories to use.
- `update_after`: `difftime` object. Automatically update the cache if it gets older than this. Set it to `Inf` to avoid updates. Defaults to seven days.
- `packages`: Packages to query, character vector.
- `dependencies`: Which kind of dependencies to include. Works the same way as the `dependencies` argument of `utils::install.packages()`.
- `recursive`: Whether to include recursive dependencies.
- `force`: Whether to force cleanup without asking the user.

## Details

`cranlike_metadata_cache$new()` creates a new cache object. Creation does not trigger the population of the cache. It is only populated on demand, when queries are executed against it. In your package, you may want to create a cache instance in the `.onLoad()` function of the package, and store it in the package namespace. As this is a cheap operation, the package will still load fast, and then the package code can refer to the common cache object.

`cmc$list()` lists all (or the specified) packages in the cache. It returns a data frame, see the list of columns below.

`cmc$async_list()` is similar, but it is asynchronous, it returns a `deferred` object.

`cmc$deps()` returns a data frame, with the (potentially recursive) dependencies of `packages`.

`cmc$async_deps()` is the same, but it is asynchronous, it returns a `deferred` object.

`cmc$revdeps()` returns a data frame, with the (potentially recursive) reverse dependencies of `packages`.

`cmc$async_revdeps()` does the same, asynchronously, it returns an `deferred` object.

`cmc$update()` updates the the metadata (as needed) in the cache, and then returns a data frame with all packages, invisibly.

`cmc$async_update()` is similar, but it is asynchronous.

`cmc$check_update()` checks if the metadata is current, and if it is not, it updates it.

`cmc$async_check_update()` is similar, but it is asynchronous.

`cmc$summary()` lists metadata about the cache, including its location and size.

`cmc$cleanup()` deletes the cache files from the disk, and also from memory.

## Columns

The metadata data frame contains all available versions (i.e. sources and binaries) for all packages. It usually has the following columns, some might be missing on some platforms.

- `package`: Package name.
- `title`: Package title.
- `version`: Package version.
- `depends`: `Depends` field from `DESCRIPTION`, or `NA_character_`.
- `suggests`: `Suggests` field from `DESCRIPTION`, or `NA_character_`.
- `built`: `Built` field from `DESCIPTION`, if a binary package, or `NA_character_`.
- `imports`: `Imports` field from `DESCRIPTION`, or `NA_character_`.
- `archs`: `Archs` entries from `PACKAGES` files. Might be missing.
- `repodir`: The directory of the file, inside the repository.
- `platform`: This is a character vector. See `default_platforms()` for more about platform names. In practice each value of the `platform` column is either
  - `"source"` for source packages,
  - a platform string, e.g. `x86_64-apple-darwin17.0` for macOS packages compatible with macOS High Sierra or newer.
- `needscompilation`: Whether the package needs compilation.

- type: bioc or cran currently.
- target: The path of the package file inside the repository.
- mirror: URL of the CRAN/BioC mirror.
- sources: List column with URLs to one or more possible locations of the package file. For source CRAN packages, it contains URLs to the Archive directory as well, in case the package has been archived since the metadata was cached.
- filesize: Size of the file, if known, in bytes, or NA_integer_.
- sha256: The SHA256 hash of the file, if known, or NA_character_.
- deps: All package dependencies, in a data frame.
- license: Package license, might be NA for binary packages.
- linkingto: LinkingTo field from DESCRIPTION, or NA_character_.
- enhances: Enhances field from DESCRIPTION, or NA_character_.
- os_type: unix or windows for OS specific packages. Usually NA.
- priority: "optional", "recommended" or NA. (Base packages are normally not included in the list, so "base" should not appear here.)
- md5sum: MD5 sum, if available, may be NA.
- sysreqs: The SystemRequirements field, if available. This lists the required system libraries or other software for the package. This is usually available for CRAN and Bioconductor package and when it is explicitly available in the repository metadata.
- published: The time the package was published at, in GMT, POSIXct class.

The data frame contains some extra columns as well, these are for internal use only.

## Examples

```
dir.create(cache_path <- tempfile())
cmc <- cranlike_metadata_cache$new(cache_path, bioc = FALSE)
cmc$list()
cmc$list("pkgconfig")
cmc$deps("pkgconfig")
cmc$revdeps("pkgconfig", recursive = FALSE)
```

---

# cran_archive_cache *Cache for CRAN archive data*

---

## Description

This is an R6 class that implements a cache from older CRAN package versions. For a higher level interface see the functions documented with cran_archive_list().

## Details

The cache is similar to cranlike_metadata_cache and has the following layers:

- The data inside the `cran_archive_cache` object.
- Cached data in the current R session.
- An RDS file in the current session's temporary directory.
- An RDS file in the user's cache directory.

It has a synchronous and an asynchronous API.

## Usage

```
cac <- cran_archive_cache$new(
  primary_path = NULL,
  replica_path = tempfile(),
  cran_mirror = default_cran_mirror(),
  update_after = as.difftime(7, units = "days"),
)

cac$list(packages = NULL, update_after = NULL)
cac$async_list(packages = NULL, update_after = NULL)

cac$update()
cac$async_update()

cac$check_update()
cac$async_check_update()

cac$summary()

cac$cleanup(force = FALSE)
```

## Arguments

- `primary_path`: Path of the primary, user level cache. Defaults to the user level cache directory of the machine.
- `replica_path`: Path of the replica. Defaults to a temporary directory within the session temporary directory.
- `cran_mirror`: CRAN mirror to use, this takes precedence over `repos`.
- `update_after`: `difftime` object. Automatically update the cache if it gets older than this. Set it to `Inf` to avoid updates. Defaults to seven days.
- `packages`: Packages to query, character vector.
- `force`: Whether to force cleanup without asking the user.

## Details

Create a new archive cache with `cran_archive_cache$new()`. Multiple caches are independent, so e.g. if you update one of them, the other existing caches are not affected.

`cac$list()` lists the versions of the specified packages, or all packages, if none were specified. `cac$async_list()` is the same, but asynchronous.

`cac$update()` updates the cache. It always downloads the new metadata. `cac$async_update()` is the same, but asynchronous.

`cac$check_update()` updates the cache if there is a newer version available. `cac$async_check_update()` is the same, but asynchronous.

`cac$summary()` returns a summary of the archive cache, a list with entries:

- `cachepath`: path to the directory of the main archive cache,
- `current_rds`: the RDS file that stores the cache. (This file might not exist, if the cache is not downloaded yet.)
- `lockfile`: the file used for locking the cache.
- ‘timestamp: time stamp for the last update of the cache.
- `size`: size of the cache file in bytes.

`cac$cleanup()` cleans up the cache files.

## Columns

`cac$list()` returns a data frame with columns:

- `package`: package name,
- `version`: package version. This is a character vector, and not a `package_version()` object. Some older package versions are not supported by `package_version()`.
- `raw`: the raw row names from the CRAN metadata.
- `mtime`: `mtime` column from the CRAN metadata. This is usually pretty close to the release date and time of the package.
- `url`: package download URL.
- `mirror`: CRAN mirror that was used to get this data.

## Examples

```
arch <- cran_archive_cache$new()
arch$update()
arch$list()
```

`cran_archive_list` *Data about older versions of CRAN packages*

## Description

CRAN mirrors store older versions of packages in `/src/contrib/Archive`, and they also store some metadata about them in `/src/contrib/Meta/archive.rds`. pkgcache can download and cache this metadata.

## Usage

```
cran_archive_list(
  cran_mirror = default_cran_mirror(),
  update_after = as.difftime(7, units = "days"),
  packages = NULL
)

cran_archive_update(cran_mirror = default_cran_mirror())

cran_archive_cleanup(cran_mirror = default_cran_mirror(), force = FALSE)

cran_archive_summary(cran_mirror = default_cran_mirror())
```

## Arguments

| | |
|---|---|
| `cran_mirror` | CRAN mirror to use, see `default_cran_mirror()`. |
| `update_after` | `difftime` object. Automatically update the cache if it gets older than this. Set it to `Inf` to avoid updates. Defaults to seven days. |
| `packages` | Character vector. Only report these packages. |
| `force` | Force cleanup in non-interactive mode. |

## Details

`cran_archive_list()` lists all versions of all (or some) packages. It updates the cached data first, if it is older than the specified limit.

`cran_archive_update()` updates the archive cache.

`cran_archive_cleanup()` cleans up the archive cache for `cran_mirror`.

`cran_archive_summary()` prints a summary about the archive cache.

## Value

`cran_archive_list()` returns a data frame with columns:

- `package`: package name,
- `version`: package version. This is a character vector, and not a `package_version()` object. Some older package versions are not supported by `package_version()`.

- raw: the raw row names from the CRAN metadata.
- mtime: mtime column from the CRAN metadata. This is usually pretty close to the release date and time of the package.
- url: package download URL.
- mirror: CRAN mirror that was used to get this data. The output is ordered according to package names (case insensitive) and release dates.

cran_archive_update() returns all archive data in a data frame, in the same format as cran_archive_list(), invisibly.

cran_archive_cleanup() returns nothing.

cran_archive_summary() returns a named list with elements:

- cachepath: Path to the directory that contains all archive cache.
- current_rds: Path to the RDS file that contains the data for the specified cran_mirror.
- lockfile: Path to the lock file for current_rds.
- timestamp: Path to the time stamp for current_rds. NA if the cache is empty.
- size: Size of current_rds. Zero if the cache is empty.

## See Also

The cran_archive_cache class for more flexibility.

## Examples

```
cran_archive_list(packages = "readr")
```

---

# current_r_platform *R platforms*

---

## Description

R platforms

## Usage

```
current_r_platform()

current_r_platform_data()

default_platforms()
```

## Details

`current_r_platform()` detects the platform of the current R version. `current_r_platform_data()` is similar, but returns the raw data instead of a character scalar.

By default pkgcache works with source packages and binary packages for the current platform. You can change this, by providing different platform names as arguments to `cranlike_metadata_cache$new()`, `repo_status()`, etc.

These functions accept the following platform names:

- `"source"` for source packages,
- `"macos"` for macOS binaries that are appropriate for the R versions pkgcache is working with. Packages for incompatible CPU architectures are dropped (defaulting to the CPU of the current macOS machine and x86_64 on non-macOS systems). The macOS Darwin version is selected based on the CRAN macOS binaries. E.g. on R 3.5.0 macOS binaries are built for macOS El Capitan.
- `"windows"` for Windows binaries for the default CRAN architecture. This is currently Windows Vista for all supported R versions, but it might change in the future. The actual binary packages in the repository might support both 32 bit and 64 builds, or only one of them. In practice 32-bit only packages are very rare. CRAN builds before and including R 4.1 have both architectures, from R 4.2 they are 64 bit only. `"windows"` is an alias to `i386+x86_64-w64-mingw32` currently.
- A platform string like `R.version$platform`, but on Linux the name and version of the distribution are also included. Examples:
  - `x86_64-apple-darwin17.0`: macOS High Sierra.
  - `aarch64-apple-darwin20`: macOS Big Sur on arm64.
  - `x86_64-w64-mingw32`: 64 bit Windows.
  - `i386-w64-mingw32`: 32 bit Windows.
  - `i386+x86_64-w64-mingw32`: 64 bit + 32 bit Windows.
  - `i386-pc-solaris2.10`: 32 bit Solaris. (Some broken 64 Solaris builds might have the same platform string, unfortunately.)
  - `x86_64-pc-linux-gnu-debian-10`: Debian Linux 10 on x86_64.
  - `x86_64-pc-linux-musl-alpine-3.14.1`: Alpine Linux.
  - `x86_64-pc-linux-gnu-unknown`: Unknown Linux Distribution on x86_64.
  - `s390x-ibm-linux-gnu-ubuntu-20.04`: Ubuntu Linux 20.04 on S390x.
  - `amd64-portbld-freebsd12.1`: FreeBSD 12.1 on x86_64.

`default_platfoms()` returns the default platforms for the current R session. These typically consist of the detected platform of the current R session, and `"source"`, for source packages.

## Value

`current_r_platform()` returns a character scalar.

`current_r_platform_data()` returns a data frame with character scalar columns:

- `cpu`,
- `vendor`,

- os,
- distribution (only on Linux),
- release (only on Linux),
- platform: the concatenation of the other columns, separated by a dash.

default_platforms() returns a character vector of the default platforms.

## Examples

```
current_r_platform()
default_platforms()
```

---

default_cran_mirror *Query the default CRAN repository for this session*

---

## Description

If options("repos") (see options()) contains an entry called "CRAN", then that is returned. If it is a list, it is converted to a character vector.

## Usage

```
default_cran_mirror()
```

## Details

Otherwise the RStudio CRAN mirror is used.

## Value

A named character vector of length one, where the name is "CRAN".

## Examples

```
default_cran_mirror()
```

## get_cranlike_metadata_cache

*The R6 object that implements the global metadata cache*

### Description

This is used by the `meta_cache_deps()`, `meta_cache_list()`, etc. functions.

### Usage

```
get_cranlike_metadata_cache()
```

### Examples

```
get_cranlike_metadata_cache()
get_cranlike_metadata_cache()$list("cli")
```

## get_graphics_api_version

*Query the version of the graphics API*

### Description

A package compiled with a certain version of the graphics API will not work with R installations that use a different version.

### Usage

```
get_graphics_api_version()
```

### Value

An integer scalar, the version of the graphics API of this R version.

### Examples

```
get_graphics_api_version()
```

## get_internals_id *Query UUID identifying the version of the R API*

### Description

Packages need to be recompiled if this id changes.

### Usage

```
get_internals_id()
```

### Value

String, a UUID.

### Examples

```
get_internals_id()
```

## meta_cache_deps *Query CRAN(like) package data*

### Description

It uses CRAN and BioConductor packages, for the current platform and R version, from the default repositories.

### Usage

```
meta_cache_deps(packages, dependencies = NA, recursive = TRUE)

meta_cache_revdeps(packages, dependencies = NA, recursive = TRUE)

meta_cache_update()

meta_cache_list(packages = NULL)

meta_cache_cleanup(force = FALSE)

meta_cache_summary()
```

### Arguments

| | |
|---|---|
| packages | Packages to query. |
| dependencies | Dependency types to query. See the dependencies parameter of utils::install.packages(). |
| recursive | Whether to query recursive dependencies. |
| force | Whether to force cleanup without asking the user. |

## Details

`meta_cache_list()` lists all packages.

`meta_cache_update()` updates all metadata. Note that metadata is automatically updated if it is older than seven days.

`meta_cache_deps()` queries packages dependencies.

`meta_cache_revdeps()` queries reverse package dependencies.

`meta_cache_summary()` lists data about the cache, including its location and size.

`meta_cache_cleanup()` deletes the cache files from the disk.

## Value

A data frame of the dependencies. For `meta_cache_deps()` and `meta_cache_revdeps()` it includes the queried `packages` as well.

## Examples

```
meta_cache_list("pkgdown")
meta_cache_deps("pkgdown", recursive = FALSE)
meta_cache_revdeps("pkgdown", recursive = FALSE)
```

---

# package_cache *A simple package cache*

---

## Description

This is an R6 class that implements a concurrency safe package cache.

## Details

By default these fields are included for every package:

- `fullpath` Full package path.
- `path` Package path, within the repository.
- `package` Package name.
- `url` URL it was downloaded from.
- `etag` ETag for the last download, from the given URL.
- `sha256` SHA256 hash of the file.

Additional fields can be added as needed.

For a simple API to a session-wide instance of this class, see `pkg_cache_summary()` and the other functions listed there.

## Usage

```
pc <- package_cache$new(path = NULL)

pc$list()
pc$find(..., .list = NULL)
pc$copy_to(..., .list = NULL)
pc$add(file, path, sha256 = shasum256(file), ..., .list = NULL)
pc$add_url(url, path, ..., .list = NULL, on_progress = NULL,
  http_headers = NULL)
pc$async_add_url(url, path, ..., .list = NULL, on_progress = NULL,
  http_headers = NULL)
pc$copy_or_add(target, urls, path, sha256 = NULL, ..., .list = NULL,
               on_progress = NULL, http_headers = NULL)
pc$async_copy_or_add(target, urls, path, ..., sha256 = NULL, ...,
               .list = NULL, on_progress = NULL, http_headers = NULL)
pc$update_or_add(target, urls, path, ..., .list = NULL,
               on_progress = NULL, http_headers = NULL)
pc$async_update_or_add(target, urls, path, ..., .list = NULL,
               on_progress = NULL, http_headers = NULL)
pc$delete(..., .list = NULL)
```

## Arguments

- `path`: For `package_cache$new()` the location of the cache. For other functions the location of the file inside the cache.
- `...`: Extra attributes to search for. They have to be named.
- `.list`: Extra attributes to search for, they have to in a named list.
- `file`: Path to the file to add.
- `url`: URL attribute. This is used to update the file, if requested.
- `sha256`: SHA256 hash of the file.
- `on_progress`: Callback to create progress bar. Passed to internal function `http_get()`.
- `target`: Path to copy the (first) to hit to.
- `urls`: Character vector or URLs to try to download the file from.
- `http_headers`: HTTP headers to add to all HTTP queries.

## Details

`package_cache$new()` attaches to the cache at `path`. (By default a platform dependent user level cache directory.) If the cache does not exists, it creates it.

`pc$list()` lists all files in the cache, returns a data frame with all the default columns, and potentially extra columns as well.

`pc$find()` list all files that match the specified criteria (`fullpath`, `path`, `package`, etc.). Custom columns can be searched for as well.

`pc$copy_to()` will copy the first matching file from the cache to `target`. It returns the data frame of *all* matching records, invisibly. If no file matches, it returns an empty (zero-row) data frame.

`pc$add()` adds a file to the cache.

`pc$add_url()` downloads a file and adds it to the cache.

`pc$async_add_url()` is the same, but it is asynchronous.

`pc$copy_or_add()` works like `pc$copy_to()`, but if the file is not in the cache, it tries to download it from one of the specified URLs first.

`pc$async_copy_or_add()` is the same, but asynchronous.

`pc$update_or_add()` is like `pc$copy_to_add()`, but if the file is in the cache it tries to update it from the urls, using the stored ETag to avoid unnecessary downloads.

`pc$async_update_or_add()` is the same, but it is asynchronous.

`pc$delete()` deletes the file(s) from the cache.

## Examples

```
## Although package_cache usually stores packages, it may store
## arbitrary files, that can be search by metadata
pc <- package_cache$new(path = tempfile())
pc$list()

cat("foo\n", file = f1 <- tempfile())
cat("bar\n", file = f2 <- tempfile())
pc$add(f1, "/f1")
pc$add(f2, "/f2")
pc$list()
pc$find(path = "/f1")
pc$copy_to(target = f3 <- tempfile(), path = "/f1")
readLines(f3)
```

---

# parse_installed *List metadata of installed packages*

---

## Description

This function is similar to `utils::installed.packages()`. See the differences below.

## Usage

```
parse_installed(
  library = .libPaths(),
  priority = NULL,
  lowercase = FALSE,
  reencode = TRUE,
  packages = NULL
)
```

## Arguments

| | |
|---|---|
| library | Character vector of library paths. |
| priority | If not NULL then it may be a "base" "recommended" NA or a vector of these to select *base* packages, *recommended* packages or *other* packages. (These are the official, CRAN supported package priorities, but you may introduce others in non-CRAN packages.) |
| lowercase | Whether to convert keys in DESCRIPTION to lowercase. |
| reencode | Whether to re-encode strings in UTF-8, from the encodings specified in the DESCRIPTION files. Re-encoding is somewhat costly, and sometimes it is not important (e.g. when you only want to extract the dependencies of the installed packages). |
| packages | If not NULL, then it must be a character vector, and only these packages will be listed. |

## Details

Differences with utils::installed.packages():

- parse_installed() cannot subset the extracted fields. (But you can subset the result.)
- parse_installed() does not cache the results.
- parse_installed() handles errors better. See Section 'Errors' below. #' * parse_installed() uses the DESCRIPTION files in the installed packages instead of the Meta/package.rds files. This should not matter, but because of a bug Meta/package.rds might contain the wrong Archs field on multi-arch platforms.
- parse_installed() reads *all* fields from the DESCRIPTION files. utils::installed.packages() only reads the specified fields.
- parse_installed() converts its output to UTF-8 encoding, from the encodings declared in the DESCRIPTION files.
- parse_installed() is considerably faster.

**Encodings:**

parse_installed() always returns its result in UTF-8 encoding. It uses the Encoding fields in the DESCRIPTION files to learn their encodings. parse_installed() does not check that an UTF-8 file has a valid encoding. If it fails to convert a string to UTF-8 from another declared encoding, then it leaves it as "bytes" encoded, without a warning.

**Errors:**

pkgcache silently ignores files and directories inside the library directory.

The result also omits broken package installations. These include

- packages with invalid DESCRIPTION files, and
- packages the current user have no access to.

These errors are reported via a condition with class pkgcache_broken_install. The condition has an errors entry, which is a data frame with columns

- file: path to the DESCRIPTION file of the broken package,
- error: error message for this particular failure.

If you intend to handle broken package installation, you need to catch this condition with withCallingHandlers().

parse_packages *Parse a repository metadata* PACAKGES* *file*

## Description

Parse a repository metadata PACAKGES* file

## Usage

```
parse_packages(path, type = NULL)
```

## Arguments

| | |
|---|---|
| path | Path to the PACKAGES* file. |
| type | Type of the file. By default it is determined automatically. Types:<br>• uncompressed,<br>• gzip compressed,<br>• bzip2 compressed,<br>• xz compressed.<br>• rds, an RDS file, which will be read using base::readRDS(). |

## Details

Non-existent, unreadable or corrupt PACKAGES files with trigger an error.
PACKAGES* files do not usually declare an encoding, but nevertheless parse_packages() works correctly if they do.

## Value

A data frame, with all columns from the file at path.

## Note

parse_packages() cannot currently read files that have very many different fields (many columns in the result data frame). The current limit is 1000. Typical PACKAGES files contain less than 20 field types.

pkg_cache_summary *Functions to query and manipulate the package cache*

## Description

`pkg_cache_summary()` returns a short summary of the state of the cache, e.g. the number of files and their total size. It returns a named list.

## Usage

```
pkg_cache_summary(cachepath = NULL)

pkg_cache_list(cachepath = NULL)

pkg_cache_find(cachepath = NULL, ...)

pkg_cache_get_file(cachepath = NULL, target, ...)

pkg_cache_delete_files(cachepath = NULL, ...)

pkg_cache_add_file(cachepath = NULL, file, relpath = dirname(file), ...)
```

## Arguments

| | |
|---|---|
| `cachepath` | Path of the cache. By default the cache directory is in `pkgcache`, within the user's cache directory. See `tools::R_user_dir()`. |
| `...` | Extra named arguments to select the package file. |
| `target` | Path where the selected file is copied. |
| `file` | File to add. |
| `relpath` | The relative path of the file within the cache. |

## See Also

The package_cache R6 class for a more flexible API.

## Examples

```
pkg_cache_summary()
pkg_cache_list()
pkg_cache_find(package = "forecast")
tmp <- tempfile()
pkg_cache_get_file(target = tmp, package = "forecast", version = "8.10")
pkg_cache_delete_files(package = "forecast")
```

## ppm_has_binaries *Does PPM build binary packages for the current platform?*

### Description

Does PPM build binary packages for the current platform?

### Usage

```
ppm_has_binaries()
```

### Value

TRUE or FALSE.

### See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/dev/.

Other PPM functions: ppm_platforms(), ppm_r_versions(), ppm_repo_url(), ppm_snapshots()

### Examples

```
current_r_platform()
ppm_has_binaries()
```

## ppm_platforms *List all platforms supported by Posit Package Manager (PPM)*

### Description

List all platforms supported by Posit Package Manager (PPM)

### Usage

```
ppm_platforms()
```

## Value

Data frame with columns:

- `name`: platform name, this is essentially an identifier,
- `os`: operating system, `linux`, `windows` or `macOS` currently,
- `binary_url`: the URL segment of the binary repository URL of this platform, see `ppm_snapshots()`.
- `distribution`: for Linux platforms the name of the distribution,
- `release`: for Linux platforms, the name of the release,
- `binaries`: whether PPM builds binaries for this platform.
- `platforms`: a list column of character vectors; for each row they list all possible matching distribution and release strings. Each string can be a fixes strings, but if it starts and ends with a forward string, then it is used as regular expression.

## See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/dev/.

Other PPM functions: `ppm_has_binaries()`, `ppm_r_versions()`, `ppm_repo_url()`, `ppm_snapshots()`

## Examples

```
ppm_platforms()
```

---

`ppm_repo_url` *Returns the current Posit Package Manager (PPM) repository URL*

---

## Description

Returns the current Posit Package Manager (PPM) repository URL

## Usage

```
ppm_repo_url()
```

## Details

This URL has the form `{base}/{repo}`, e.g. `https://packagemanager.posit.co/all`.

To configure a hosted PPM instance, set the `PKGCACHE_PPM_URL` environment variable to the base URL (e.g. `https://packagemanager.posit.co`).

To use `repo_add()` with PPM snapshots, you may also set the `PKGCACHE_PPM_REPO` environment variable to the name of the default repository.

On Linux, instead of setting these environment variables, you can also add a PPM repository to the `repos` option, see `base::options()`. If the environment variables are not set, then `ppm_repo_url()` will try to extract the PPM base URL and repository name from this option.

If the PKGCACHE_PPM_URL environment variable is not set, and the repos option does not contain a PPM URL (on Linux), then pkgcache uses the public PPM instance at https://packagemanager.posit.co, with the cran repository.

## Value

String scalar, the repository URL of the configured PPM instance. If no PPM instance is configured, then the URL of the Posit Public Package Manager instance. It includes the repository name, e.g. https://packagemanager.posit.co/all.

## See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/dev/.

repo_resolve() and repo_add() to find and configure PPM snapshots.

Other PPM functions: ppm_has_binaries(), ppm_platforms(), ppm_r_versions(), ppm_snapshots()

## Examples

```
ppm_repo_url()
```

---

# ppm_r_versions *List all R versions supported by Posit Package Manager (PPM)*

---

## Description

List all R versions supported by Posit Package Manager (PPM)

## Usage

```
ppm_r_versions()
```

## Value

Data frame with columns:

- r_version: minor R versions, i.e. version numbers containing the first two components of R versions supported by this PPM instance.

## See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/dev/.

Other PPM functions: ppm_has_binaries(), ppm_platforms(), ppm_repo_url(), ppm_snapshots()

## Examples

```
ppm_r_versions()
```

ppm_snapshots *List all available Posit Package Manager (PPM) snapshots*

## Description

List all available Posit Package Manager (PPM) snapshots

## Usage

```
ppm_snapshots()
```

## Details

The repository URL of a snapshot has the following form on Windows:

```
{base}/{repo}/{id}
```

where `{base}` is the base URL for PPM (see `ppm_repo_url()`) and `{id}` is either the date or id of the snapshot, or `latest` for the latest snapshot. E.g. these are equivalent:

```
https://packagemanager.posit.co/cran/5
https://packagemanager.posit.co/cran/2017-10-10
```

On a Linux distribution that has PPM support, the repository URL that contains the binary packages looks like this:

```
{base}/{repo}/__linux__/{binary_url}/{id}
```

where `{id}` is as before, and `{binary_url}` is a code name for a release of a supported Linux distribution. See the `binary_url` column of the result of `ppm_platforms()` for these code names.

## Value

Data frame with two columns:

- `date`: the time the snapshot was taken, a `POSIXct` vector,
- `id`: integer id of the snapshot, this can be used in the repository URL.

## See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/dev/.

Other PPM functions: `ppm_has_binaries()`, `ppm_platforms()`, `ppm_r_versions()`, `ppm_repo_url()`

## Examples

```
ppm_snapshots()
```

repo_auth *Authenticated repositories*

## Description

`repo_auth()` lists authentication information for all configured repositories.

## Usage

```
repo_auth(
  r_version = getRversion(),
  bioc = TRUE,
  cran_mirror = default_cran_mirror(),
  check_credentials = TRUE
)
```

## Arguments

| | |
|---|---|
| `r_version` | R version(s) to use for the Bioconductor repositories, if `bioc` is `TRUE`. |
| `bioc` | Whether to add Bioconductor repositories, even if they are not configured in the `repos` option. |
| `cran_mirror` | The CRAN mirror to use, see `default_cran_mirror()`. |
| `check_credentials` | Whether to check that credentials are available for the authenticated repositories. |

## Details

pkgcache supports HTTP basic authentication when interacting with CRAN-like repositories. To use authentication, include a username in the repo URL:

```
https://<username>@<repo-host>/<repo-path>
```

pkgcache tries to obtain the passwords for the authenticated CRAN-like repos from two sources, in this order:

1. A "netrc" file. If the `NETRC` environment variable is set, it should point to a file in "netrc" format. Otherwise pkgcache uses the `~/.netrc` file in the current user's home directory, if it exists. On Windows it also tries the `~/_netrc` file.
2. The system's credential store, via the keyring package.

See the documentation of libcurl for details about the format of the netrc file.

When looking for the password in the system credential store, pkgcache looks at the following keys, in this order:

```
https://<username>@repo-host/<repo-path>
https://repo-host/<repo-path>
https://<username>@repo-host
https://repo-host
```

To add an authenticated repository use `repo_add()` with the `username` argument. Alternatively, you can set the `repos` option directly using `base::options()` and including the username in the repository URL.

## Value

Data frame with columns:

- all columns from the output of `repo_get()`,
- `auth_domains`: authentication domains. pkgcache tries to find the credentials for these domains, until the search is successful or all domains fail. This column is a list column of character vectors.
- `auth_domain`: if the credential lookup is successful, then this is the authentication domain that was used to get the credentials.
- `auth_source`: where the credentials were found. E.g. `keyring:macos` means it was in the default macos keyring.
- `auth_error`: for failed credential searches this is the description of why the search failed. E.g. maybe the keyring package is not installed, or pkgcache found no credentials for any of the authentication domains.

---

`repo_get` *Query and set the list of CRAN-like repositories*

---

## Description

pkgcache uses the `repos` option, see `options()`. It also automatically uses the current Bioconductor repositories, see `bioc_version()`. These functions help to query and manipulate the `repos` option.

## Usage

```
repo_get(
  r_version = getRversion(),
  bioc = TRUE,
  cran_mirror = default_cran_mirror()
)

repo_resolve(spec, username = NULL)

repo_add(..., .list = NULL, username = NULL)

with_repo(repos, expr)
```

## Arguments

| | |
|---|---|
| r_version | R version(s) to use for the Bioconductor repositories, if bioc is TRUE. |
| bioc | Whether to add Bioconductor repositories, even if they are not configured in the repos option. |
| cran_mirror | The CRAN mirror to use, see default_cran_mirror(). |
| spec | A single repository specification, a possibly named character scalar. See details below. |
| username | User name to set, for authenticated repositories, see repo_auth(). |
| ... | Repository specifications. See details below. |
| .list | List or character vector of repository specifications, see details below. |
| repos | A list or character vector of repository specifications. |
| expr | R expression to evaluate. |

## Details

repo_get() queries the repositories pkgcache uses. It uses the repos option (see options), and also the default Bioconductor repository.

repo_resolve() resolves a single repository specification to a repository URL.

repo_add() adds a new repository to the repos option. (To remove a repository, call option() directly, with the subset that you want to keep.)

with_repo() temporarily adds the repositories in repos, evaluates expr, and then resets the configured repositories.

## Value

repo_get() returns a data frame with columns:

- name: repository name. Names are informational only.
- url: repository URL.
- type: repository type. This is also informational, currently it can be cran for CRAN, bioc for a Bioconductor repository, and cranlike: for other repositories.
- r_version: R version that is supposed to be used with this repository. This is only set for Bioconductor repositories. It is * for others. This is also informational, and not used when retrieving the package metadata.
- bioc_version: Bioconductor version. Only set for Bioconductor repositories, and it is NA for others.
- username: user name, for authenticated repositories.
- has_password: whether repo_get() could find the password for this repository. Call repo_auth() for more information if the credential lookup failed.

repo_resolve() returns a named character vector, with the URL(s) of the repository.

repo_add() returns the same data frame as repo_get(), invisibly.

with_repo() returns the value of expr.

### Repository specifications

The format of a repository specification is a named or unnamed character scalar. If the name is missing, pkgcache adds a name automatically. The repository named `CRAN` is the main CRAN repository, but otherwise names are informational.

Currently supported repository specifications:

- URL pointing to the root of the CRAN-like repository. Example:

  ```
  https://cloud.r-project.org
  ```

- `PPM@latest`, PPM (Posit Package Manager, formerly RStudio Package Manager), the latest snapshot.
- `PPM@<date>`, PPM (Posit Package Manager, formerly RStudio Package Manager) snapshot, at the specified date.
- `PPM@<package>-<version>` PPM snapshot, for the day after the release of `<version>` of `<package>`.
- `PPM@R-<version>` PPM snapshot, for the day after R `<version>` was released.

Still works for dates starting from 2017-10-10, but now deprecated, because MRAN is discontinued:

- `MRAN@<date>`, MRAN (Microsoft R Application Network) snapshot, at the specified date.
- `MRAN@<package>-<version>` MRAN snapshot, for the day after the release of `<version>` of `<package>`.
- `MRAN@R-<version>` MRAN snapshot, for the day after R `<version>` was released.

Notes:

- See more about PPM at https://packagemanager.posit.co/client/#/.
- The `RSPM@` prefix is still supported and treated the same way as `PPM@`.
- The MRAN service is now retired, see `https://techcommunity.microsoft.com/blog/azuresqlblog/microsoft-` for details.
- `MRAN@...` repository specifications now resolve to PPM, but note that PPM snapshots are only available from 2017-10-10. See more about this at https://posit.co/blog/migrating-from-mran-to-posit-pack
- All dates (or times) can be specified in the ISO 8601 format.
- If PPM does not have a snapshot available for a date, the next available date is used.
- Dates that are before the first, or after the last PPM snapshot will trigger an error.
- Unknown R or package versions will trigger an error.

### See Also

Other repository functions: `repo_status()`

## Examples

```
repo_get()

repo_resolve("PPM@2021-01-21")
#' repo_resolve("PPM@dplyr-1.0.0")
#' repo_resolve("PPM@R-4.0.0")


with_repo(c(CRAN = "PPM@dplyr-1.0.0"), repo_get())
with_repo(c(CRAN = "PPM@dplyr-1.0.0"), meta_cache_list(package = "dplyr"))

with_repo(c(CRAN = "MRAN@2018-06-30"), summary(repo_status()))
```

---

# repo_status *Show the status of CRAN-like repositories*

---

## Description

It checks the status of the configured or supplied repositories, for the specified platforms and R versions.

## Usage

```
repo_status(
  platforms = default_platforms(),
  r_version = getRversion(),
  bioc = TRUE,
  cran_mirror = default_cran_mirror()
)
```

## Arguments

| | |
|---|---|
| platforms | Platforms to use, default is default_platforms(). |
| r_version | R version(s) to use, the default is the current R version, via getRversion(). |
| bioc | Whether to add the Bioconductor repositories. If you already configured them via options(repos), then you can set this to FALSE. See bioc_version() for the details about how pkgcache handles Bioconductor repositories. |
| cran_mirror | The CRAN mirror to use, see default_cran_mirror(). |

## Details

The returned data frame has a summary() method, which shows the same information is a concise table. See examples below.

## Value

A data frame that has a row for every repository, on every queried platform and R version. It has these columns:

- `name`: the name of the repository. This comes from the names of the configured repositories in `options("repos")`, or added by pkgcache. It is typically `CRAN` for CRAN, and the current Bioconductor repositories are `BioCsoft`, `BioCann`, `BioCexp`, `BioCworkflows`, `BioCbooks`.
- `url`: base URL of the repository.
- `bioc_version`: Bioconductor version, or `NA` for non-Bioconductor repositories.
- `username`: Included if at least one repository is authenticated. `NA_character_` for repositories without authentication. See `repo_auth()`.
- `has_password`: `TRUE` is the function could retrieve the password for the authenticated repository. It is `NA` for repositories without authentication. This column is included only if at least one repository has authentication. See `repo_auth()`.
- `platform`: platform, see `default_platforms()` for possible values.
- `path`: the path to the packages within the base URL, for a given platform and R version.
- `r_version`: R version, one of the specified R versions.
- `ok`: Logical flag, whether the repository contains a metadata file for the given platform and R version.
- `ping`: HTTP response time of the repository in seconds. If the `ok` column is `FALSE`, then this columns in `NA`.
- `error`: the error object if the HTTP query failed for this repository, platform and R version.

## See Also

Other repository functions: `repo_get()`

## Examples

```
repo_status()
rst <- repo_status(
  platforms = c("windows", "macos"),
  r_version = c("4.0", "4.1")
)
summary(rst)
```

# Index