Package 'pkgdepends'

December 17, 2022

Title Package Dependency Resolution and Downloads

Version 0.4.0

Description Find recursive dependencies of 'R' packages from various sources. Solve the dependencies to obtain a consistent set of packages to install. Download packages, and install them. It supports packages on 'CRAN', 'Bioconductor' and other 'CRAN-like' repositories, 'GitHub', package 'URLs', and local package trees and files. It caches metadata and package files via the 'pkgcache' package, and performs all 'HTTP' requests, downloads, builds and installations in parallel. 'pkgdepends' is the workhorse of the 'pak' package.

License MIT + file LICENSE

URL https://github.com/r-lib/pkgdepends#readme

BugReports https://github.com/r-lib/pkgdepends/issues

Depends R (>= 3.4)

Imports callr (>= 3.3.1), cli (>= 3.4.0), curl, desc (>= 1.2.0), filelock (>= 1.0.2), glue, jsonlite, lpSolve, pkgbuild (>= 1.0.2), pkgcache (>= 2.0.0), prettyunits (>= 1.1.1), processx (>= 3.4.2), ps, R6, rprojroot, stats, utils, withr (>= 2.1.1), zip (>= 2.1.0)

Suggests asciicast (>= 2.2.0.9000), covr, debugme, fansi, fs, htmlwidgets, mockery, pak, pingr (>= 2.0.0), rmarkdown, rstudioapi, spelling, testthat, tibble, webfakes (>= 1.1.5.9000)

Config/Needs/coverage covr, rmarkdown, svglite

Config/Needs/dev rmarkdown, svglite

Config/Needs/website r-lib/asciicast, pkgdown (>= 2.0.2), tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.2.1.9000

NeedsCompilation no

Author Gábor Csárdi [aut, cre],
RStudio [cph, fnd]

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

Repository CRAN

Date/Publication 2022-12-17 09:30:02 UTC

R topics documented:

pkgdepends-package . . . 2
as_pkg_dependencies . . . 5
current_r_platform . . . 6
install_package_plan . . . 8
install_plans . . . 8
is_valid_package_name . . . 10
lib_status . . . 10
new_pkg_deps . . . 12
new_pkg_download_proposal . . . 17
new_pkg_installation_plan . . . 21
new_pkg_installation_proposal . . . 24
parse_pkg_refs . . . 33
pkg_config . . . 34
pkg_dep_types_hard . . . 36
pkg_downloads . . . 37
pkg_name_check . . . 38
pkg_refs . . . 39
pkg_resolution . . . 45
pkg_rx . . . 47
pkg_solution . . . 48

Index **50**

`pkgdepends-package` *pkgdepends: Package Dependency Resolution and Downloads*

Description

pkgdepends is a toolkit for package dependencies, downloads and installations, to be used in other packages. If you are looking for a package manager, see pak.

Features

- Look up package dependencies recursively.
- Visualize package dependencies.
- Download packages and their dependencies.
- Install downloaded packages.

- Includes a dependency solver to find a consistent set of dependencies.
- Supports CRAN and Bioconductor packages automatically.
- Supports packages on GitHub.
- Supports local package file and trees.
- Supports the `Remotes` entry in the `DESCRIPTION` file.
- Caches metadata and downloaded packages via pkgcache
- Performs all downloads and HTTP queries concurrently.
- Builds and installs packages in parallel.

Install

Once on CRAN, install the package with:

```
install.packages("pkgdepends")
```

Usage

```
library(pkgdepends)
```

Package references:

A package reference (ref) specifies a location from which an R package can be obtained from. Examples:

```
devtools
cran::devtools
bioc::Biobase
r-lib/pkgdepends
https://github.com/r-lib/pkgdepends
local::~/works/shiny
```

See "Package references" for details.

Package dependencies:

Dependencies of the development version of the cli package:

```
pd <- new_pkg_deps("r-lib/pkgcache")
pd$solve()
pd$draw()
```

See the `pkg_deps` class for details.

Package downloads:

Downloading all dependencies of a package:

```
pdl <- new_pkg_download_proposal("r-lib/cli")
pdl$resolve()
pdl$download()
```

See the `pkg_download_proposal` class for details.

Package installation:

Installing or updating a set of package:

```
lib <- tempfile()
dir.create(lib)
pdi <- new_pkg_installation_proposal(
  "r-lib/cli",
  config = list(library = lib)
)
pdi$solve()
pdi$download()
pdi$install()
```

Dependency resolution:

`pkg_deps`, `pkg_download_proposal` and `pkg_installation_proposal` all resolve their dependencies recursively, to obtain information about all packages needed for the specified package references. See "Dependency resolution" for details.

The dependency solver:

The dependency solver takes the resolution information, and works out the exact versions of each package that must be installed, such that version and other requirements are satisfied. See "The dependency solver" for details.

Installation plans:

`pkg_installation_proposal` can create installation plans, and then also install them. It is also possible to import installation plans that were created by other tools. See "Installation plans" for details.

Configuration:

The details of `pkg_deps`, `pkg_download_proposal` and `pkg_installation_proposal` can be tuned with a list of configuration options. See "Configuration" for details.

Related

- pak – R package manager
- pkgcache – Metadata and package cache
- devtools – Tools for R package developers

Code of Conduct:

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

License

Author(s)

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

Other contributors:

- RStudio [copyright holder, funder]

See Also

Useful links:

- https://github.com/r-lib/pkgdepends#readme
- Report bugs at https://github.com/r-lib/pkgdepends/issues

---

as_pkg_dependencies *Shorthands for dependency specifications*

Description

Shorthands for dependency specifications

Usage

```
as_pkg_dependencies(deps)
```

Arguments

deps	See below.

Details

R packages may have various types of dependencies, see Writing R Extensions.

pkgdepends groups dependencies into three groups:

- hard dependencies: "Depends", "Imports", and "LinkingTo",
- soft dependencies: "Suggests" and "Enhances",
- extra dependencies, see below.

pkgdepends supports concise ways of specifying which types of dependencies of a package should be installed. It is similar to how `utils::install.packages()` interprets its `dependencies` argument.

You typically use one of these values:

- `NA` or `"hard"` to install a package and its required dependencies,
- `TRUE` to install all required dependencies, plus optional and development dependencies.

If you need more flexibility, the full description of possible values for the `deps` argument are:

- TRUE: This means all hard dependencies plus Suggests for direct installations, and hard dependencies only for dependent packages.
- FALSE: no dependencies are installed at all.
- NA (any atomic type, so NA_character_, etc. as well): only hard dependencies are installed. See pkg_dep_types_hard().
- If a list with two entries named direct and indirect, it is taken as the requested dependency types, for direct installations and dependent packages.
- If a character vector, then it is taken as the dependency types for direct installations, and the hard dependencies are used for the dependent packages.

If "hard" is included in the value or a list element, then it is replaced by the hard dependency types. If "soft" or "all" is included, then it is replaced by all hard and soft dependency.

Extra dependencies:

pkgdepends supports extra dependency types for direct installations not from CRAN-like repositories. These are specified with a Config/Needs/ prefix in the DESCRIPTION and they can contain package references, separated by commas. For example you can specify packages that are only needed for the pkgdown website of the package:

```
Config/Needs/website: r-lib/pkgdown
```

To use these dependency types, you need to specify them in the deps argument to pkgdepends functions.

Note that Config/Needs/* fields are currently *not* used from CRAN packages, and packages in CRAN-like repositories in general.

Usually you specify that a Config/Needs/* dependency type should be installed together with "hard" or "all", to install all hard or soft dependencies as well.

Value

A named list with two character vectors: direct, indirect, the dependency types to use for direct installations and dependent packages.

See Also

Other package dependency utilities: pkg_dep_types_hard()

---

current_r_platform *R platforms*

Description

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.

Usage

```
current_r_platform()

default_platforms()
```

Details

`current_r_platform()` detects the platform of the current R version.

By default pkgdepends works with source packages and binary packages for the current platform. You can change this, see 'Configuration'.

The following platform names can be configured and returned by `current_r_platform()` and `default_platforms()`:

- `"source"` for source packages,
- 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.

In addition, the following platform names can be used to configure pkgdepends:

- `"macos"` for macOS binaries that are appropriate for the R versions pkgdepends is working with (defaulting to the version of the current session), as defined by CRAN 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.

Value

`current_r_platform()` returns a string, the name of the current platform.

`default_platforms()` returns a character vector of platform names.

Examples

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

`install_package_plan` *Perform a package installation plan*

Description

See 'Installation plans' for the details and the format.

Usage

```
install_package_plan(
  plan,
  lib = .libPaths()[[1]],
  num_workers = 1,
  cache = NULL
)
```

Arguments

`plan`	Package plan object, a data frame, see 'Installation plans' for the format.
`lib`	Library directory to install to.
`num_workers`	Number of worker processes to use.
`cache`	Package cache to use, or `NULL`.

Value

Information about the installation process.

`install_plans` *Installation plans*

Description

An installation plan contains all data that is needed to install a set of package files. It is usually created from an installation proposal with solving the dependencies and downloading the package files.

Details

It is also possible to create an installation plan a different way. An installation plan object must be a data frame, with at least the following columns:

- `package`: The name of the package.
- `type`: The type of the package reference.
- `binary`: Whether the package is a binary package.
- `file`: Full path to the package file or directory.
- `dependencies`: A list column that lists the names of the dependent packages for each package.
- `needscompilation`: Whether the package needs compilation. This should be `FALSE` for binary packages.

For installation plans created via pkg_installation_proposal, the plan contains all columns from `pkg_download_result` objects, and some additional ones:

- `library`: the library the package is supposed to be installed to.
- `direct`: whether the package was directly requested or it is installed as a dependency.
- vignettes: whether the vignettes need to be (re)built.
- `packaged`: whether `R CMD build` was already called for the package.

See Also

pkg_installation_proposal to create install plans, `install_package_plan()` to install plans from any source.

Examples

```
## Not run:
pdi <- new_pkg_installation_proposal(
  "pak",
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$download()
pdi$get_install_plan()

## End(Not run)
```

is_valid_package_name *Check whether a package name is valid*

Description

Check whether a package name is valid

Usage

```
is_valid_package_name(nm)
```

Arguments

nm	Potential package name, string of length 1.

Value

Logical flag. If FALSE, then the reason attribute contains a character string, the explanation why the package name is invalid. See examples below.

Examples

```
is_valid_package_name("pak")
is_valid_package_name("pkg")
is_valid_package_name("pak\u00e1ge")
is_valid_package_name("good-package")
is_valid_package_name("x")
is_valid_package_name("1stpackage")
is_valid_package_name("dots.")
```

lib_status *Status of packages in a library*

Description

Query data of all packages in a package library.

Usage

```
lib_status(library = .libPaths()[1], packages = NULL)
```

Arguments

library	Path to library.
packages	If not NULL, then only these packages are shown.

Value

Data frame that contains data about the packages installed in the library.

It has always has columns:

- `biocviews`: the corresponding field from `DESCRIPTION`, it must be present for all Bioconductor packages, other packages typically don't have it.
- `built`: the `Built` field from `DESCRIPTION`.
- `depends`, `suggests`, `Imports`, `linkingto`, `enhances`: the corresponding fields from the `DESCRIPTION` files.
- `deps`: A list or data frames, the dependencies of the package. It has columns: `ref`, `type` (dependency type in lowercase), `package` (dependent package, or `R`), `op` and `version`, for last two are for version requirement. `op` can be `>=`, `>`, `==` or `<=`, although the only the first one is common in practice.
- `library`: path to the package library containing the package.
- `license`: from `DESCRIPTION`.
- `md5sum`: from `DESCTIPTION`, typically `NA`, except on Windows.
- `needscompilation`: from `DESCRIPTION`, this column is logical.
- `package`: package name.
- `platform`: from the `Built` field in `DESCRIPTION`, the current platform if missing from `DESCRIPTION`.
- `priority`: from `DESCRIPTION`, usually `base`, `recommended`, or missing.
- `ref`: the corresponding `installed::*` package reference.
- `repository`: from `DESCRIPTION`. For packages from a CRAN repository this is `CRAN`, some other repositories, e.g. R-universe adds the repository URL here.
- `repotype`: `cran`, `bioc` or missing.
- `rversion`: from the `Built` field. If no such field, then the current R version.
- `sysreqs`: the `SystemRequirements` field from `DESCRIPTION`.
- `title`: package title.
- `type`: always `installed`.
- `version`: package version (as string).

Most of these columns are unchanged from `DESCRIPTION`, but pkgdepends also adds a couple.

Notes::

- In addition, it also has all `remote*` and `config/needs/*` entries from the `DESCRIPTION` files. (Case insensitive.)
- All columns are of type `character`, except for `needscompilation`, which is logical and `deps`, which is a list columns.
- If an entry is missing for a package, it is set to `NA`.
- Note that column names are lowercase, even if the corresponding entries are not in `DESCRIPTION`.
- The order of the columns is not deterministic, so don't assume any order.
- Additional columns might be present, these are internal for pkgdepends and should not be used in user code.

new_pkg_deps *R6 class for package dependency lookup*

Description

Look up dependencies of R packages from various sources.

Usage

```
new_pkg_deps(refs, ...)
```

Arguments

refs	Package names or references. See 'Package references' for the syntax.
...	Additional arguments, passed to pkg_deps$new().

Details

new_pkg_deps() creates a new object from the pkg_deps class. The advantage of new_pkg_deps() compared to using the pkg_deps constructor directly is that it avoids making pkgdepends a build time dependency.

The usual steps to query package dependencies are:

1. Create a pkg_deps object with new_pkg_deps().
2. Resolve all possible dependencies with pkg_deps$resolve().
3. Solve the dependencies, to obtain a subset of all possible dependencies that can be installed together, with pkg_deps$solve().
4. Call pkg_deps$get_solution() to list the result of the dependency solver.

Value

new_pkg_deps() returns a new pkg_deps object.

Methods

Public methods:

- pkg_deps$new()
- pkg_deps$get_refs()
- pkg_deps$get_config()
- pkg_deps$resolve()
- pkg_deps$async_resolve()
- pkg_deps$get_resolution()
- pkg_deps$get_solve_policy()
- pkg_deps$set_solve_policy()
- pkg_deps$solve()

- pkg_deps$get_solution()
- pkg_deps$stop_for_solution_error()
- pkg_deps$draw()
- pkg_deps$format()
- pkg_deps$print()
- pkg_deps$clone()

Method `new()`**:** Create a new pkg_deps object. Consider using `new_pkg_deps()` instead of calling the constructor directly.

The returned object can be used to look up (recursive) dependencies of R packages from various sources. To perform the actual lookup, you'll need to call the `resolve()` method.

Usage:

```
pkg_deps$new(
  refs,
  config = list(),
  policy = c("lazy", "upgrade"),
  remote_types = NULL
)
```

Arguments:

`refs` Package names or references. See 'Package references' for the syntax.

`config` Configuration options, a named list. See 'Configuration'.

`policy` Solution policy. See 'The dependency solver'.

`remote_types` Custom remote ref types, this is for advanced use, and experimental currently.

Returns: A new pkg_deps object.

Method `get_refs()`**:** The package refs that were used to create the pkg_deps object.

Usage:

```
pkg_deps$get_refs()
```

Returns: A character vector of package refs that were used to create the pkg_deps object.

Method `get_config()`**:** Configuration options for the pkg_deps object. See 'Configuration' for details.

Usage:

```
pkg_deps$get_config()
```

Returns: See 'Configuration' for the configuration entries.

Method `resolve()`**:** Resolve the dependencies of the specified package references. This usually means downloading metadata from CRAN and Bioconductor, unless already cached, and also from GitHub if GitHub refs were included, either directly or indirectly. See 'Dependency resolution' for details.

Usage:

```
pkg_deps$resolve()
```

Returns: The pkg_deps object itself, invisibly.

Method `async_resolve()`**:** The same as `resolve()`, but asynchronous. This method is for advanced use.

Usage:

```
pkg_deps$async_resolve()
```

Returns: A deferred value.

Method `get_resolution()`**:** Query the result of the dependency resolution. This method can be called after `resolve()` has completed.

Usage:

```
pkg_deps$get_resolution()
```

Returns: A pkg_resolution_result object, which is also a data frame. See 'Dependency resolution' for its columns.

Method `get_solve_policy()`**:** Returns the current policy of the dependency solver. See 'The dependency solver' for details.

Usage:

```
pkg_deps$get_solve_policy()
```

Returns: A character vector of length one.

Method `set_solve_policy()`**:** Set the current policy of the dependency solver. If the object already contains a solution and the new policy is different than the old policy, then the solution is deleted. See 'The dependency solver' for details.

Usage:

```
pkg_deps$set_solve_policy(policy = c("lazy", "upgrade"))
```

Arguments:

`policy` Policy to set.

Method `solve()`**:** Solve the package dependencies. Out of the resolved dependencies, it works out a set of packages, that can be installed together to create a functional installation. The set includes all directly specified packages, and all required (or suggested, depending on the configuration) packages as well. It includes every package at most once. See 'The dependency solver' for details.

`solve()` calls `resolve()` automatically, if it hasn't been called yet.

Usage:

```
pkg_deps$solve()
```

Returns: The `pkg_deps` object itself, invisibly.

Method `get_solution()`**:** Returns the solution of the package dependencies.

Usage:

```
pkg_deps$get_solution()
```

Returns: A pkg_solution_result object, which is a list. See pkg_solution_result for details.

Method `stop_for_solution_error()`**:** Error if the dependency solver failed to find a consistent set of packages that can be installed together.

Usage:

```
pkg_deps$stop_for_solution_error()
```

Method `draw()`**:** Draw a tree of package dependencies. It returns a `tree` object, see `cli::tree()`. Printing this object prints the dependency tree to the screen.

Usage:

```
pkg_deps$draw()
```

Returns: A `tree` object from the cli package, see `cli::tree()`.

Method `format()`**:** Format a `pkg_deps` object, typically for printing.

Usage:

```
pkg_deps$format(...)
```

Arguments:

`...` Not used currently.

Returns: A character vector, each element should be a line in the printout.

Method `print()`**:** Prints a `pkg_deps` object to the screen. The printout includes:

- The package refs.
- Whether the object has the resolved dependencies.
- Whether the resolution had errors.
- Whether the object has the solved dependencies.
- Whether the solution had errors.
- Advice on which methods to call next.

See the example below.

Usage:

```
pkg_deps$print(...)
```

Arguments:

`...` not used currently.

Returns: The `pkg_deps` object itself, invisibly.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
pkg_deps$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Method initialize()
pd <- pkg_deps$new("r-lib/pkgdepends")
pd
```

```
# Method get_refs()
pd <- new_pkg_deps(c("pak", "jsonlite"))
pd$get_refs()


# Method get_config()
pd <- new_pkg_deps("pak")
pd$get_config()


# Method resolve()
pd <- new_pkg_deps("pak")
pd$resolve()
pd$get_resolution()


# Method get_resolution()
pd <- new_pkg_deps("r-lib/pkgdepends")
pd$resolve()
pd$get_resolution()


# Method get_solve_policy()
pdi <- new_pkg_deps("r-lib/pkgdepends")
pdi$get_solve_policy()
pdi$set_solve_policy("upgrade")
pdi$get_solve_policy()


# Method set_solve_policy()
pdi <- new_pkg_deps("r-lib/pkgdepends")
pdi$get_solve_policy()
pdi$set_solve_policy("upgrade")
pdi$get_solve_policy()


# Method solve()
pd <- new_pkg_deps("r-lib/pkgdepends")
pd$resolve()
pd$solve()
pd$get_solution()


# Method get_solution()
pd <- new_pkg_deps("pkgload")
pd$resolve()
pd$solve()
pd$get_solution()


# Method stop_for_solution_error()
# This is an error, because the packages conflict:
pd <- new_pkg_deps(
```

```
  c("r-lib/pak", "cran::pak"),
  config = list(library = tempfile())
)
pd$resolve()
pd$solve()
pd
# This fails:
# pd$stop_for_solution_error()


# Method draw()
pd <- new_pkg_deps("pkgload")
pd$solve()
pd$draw()


# Method print()
pd <- new_pkg_deps("r-lib/pkgdepends")
pd

pd$resolve()
pd

pd$solve()
pd
```

`new_pkg_download_proposal` *R6 class for package downloads*

Description

Download packages with their dependencies, from various sources.

Usage

```
new_pkg_download_proposal(refs, ...)
```

Arguments

`refs`	Package names or references. See 'Package references' for the syntax.
`...`	Additional arguments, passed to `pkg_download_proposal$new()`.

Details

`new_pkg_download_proposal()` creates a new object from the `pkg_download_proposal` class, that can be used to look up and download R packages and their dependencies. The advantage

of new_pkg_download_proposal() compared to using the pkg_download_proposal constructor directly is that it avoids making pkgdepends a build time dependency.

Typical workflow to download a set of packages:

1. Create a pkg_download_proposal object with new_pkg_download_proposal().
2. Resolve all possible dependencies with pkg_download_proposal$resolve().
3. Download all files with pkg_download_proposal$download().
4. Get the data about the packages and downloads with pkg_download_proposal$get_downloads().

Value

new_pkg_download_proposal() returns a new pkg_download_proposal object.

Methods

Public methods:

- pkg_download_proposal$new()
- pkg_download_proposal$get_refs()
- pkg_download_proposal$get_config()
- pkg_download_proposal$resolve()
- pkg_download_proposal$async_resolve()
- pkg_download_proposal$get_resolution()
- pkg_download_proposal$download()
- pkg_download_proposal$async_download()
- pkg_download_proposal$get_downloads()
- pkg_download_proposal$stop_for_download_error()
- pkg_download_proposal$format()
- pkg_download_proposal$print()
- pkg_download_proposal$clone()

Method new(): Create a new pkg_download_proposal object. Consider using new_pkg_download_proposal() instead of calling the constructor directly.

The returned object can be used to look up (recursive) dependencies of R packages from various sources, and then to download the package files.

Usage:

```
pkg_download_proposal$new(refs, config = list(), remote_types = NULL)
```

Arguments:

refs Package names or references. See 'Package references' for the syntax.

config Configuration options, a named list. See 'Configuration'.

remote_types Custom remote ref types, this is for advanced use, and experimental currently.

Examples:

```
pdl <- pkg_download_proposal$new("r-lib/pkgdepends")
pdl
```

Method `get_refs()`**:** The package refs that were used to create the `pkg_download_proposal` object.

Usage:

```
pkg_download_proposal$get_refs()
```

Returns: A character vector of package refs that were used to create the `pkg_download_proposal` object.

Method `get_config()`**:** Configuration options for the `pkg_download_proposal` object. See 'Configuration' for details.

Usage:

```
pkg_download_proposal$get_config()
```

Returns: Named list. See 'Configuration' for the configuration options.

Method `resolve()`**:** Resolve the dependencies of the specified package references. This usually means downloading metadata from CRAN and Bioconductor, unless already cached, and also from GitHub if GitHub refs were included, either directly or indirectly. See 'Dependency resolution' for details.

Usage:

```
pkg_download_proposal$resolve()
```

Returns: The `pkg_download_proposal` object itself, invisibly.

Method `async_resolve()`**:** The same as `resolve()`, but asynchronous. This method is for advanced use.

Usage:

```
pkg_download_proposal$async_resolve()
```

Returns: A deferred value.

Method `get_resolution()`**:** Query the result of the dependency resolution. This method can be called after `resolve()` has completed.

Usage:

```
pkg_download_proposal$get_resolution()
```

Returns: A pkg_resolution_result object, which is also a data frame. See 'Dependency resolution' for its columns.

Method `download()`**:** Download all resolved packages. It uses the package cache in the pkgcache package by default, to avoid downloads if possible.

Usage:

```
pkg_download_proposal$download()
```

Returns: The `pkg_download_proposal` object, invisibly.

Method `async_download()`**:** The same as `download()`, but asynchronous. This method is for advanced use.

Usage:

```
pkg_download_proposal$async_download()
```

Returns: A deferred value.

Method `get_downloads()`**:** Returns the summary of the package downloads.

Usage:

```
pkg_download_proposal$get_downloads()
```

Returns: A pkg_download_result object, which is a list. See pkg_download_result for details.

Method `stop_for_download_error()`**:** Throw and error if the some of the downloads have failed for the most recent `pkg_download_proposal$download()` call.

Usage:

```
pkg_download_proposal$stop_for_download_error()
```

Method `format()`**:** Format a `pkg_download_proposal` object, typically for printing.

Usage:

```
pkg_download_proposal$format(...)
```

Arguments:

`...` not used currently.

Returns: Nothing. A character vector, each element should be a line in the printout.

Method `print()`**:** Prints a `pkg_download_proposal` object to the screen. The printout includes:

- The package refs.
- Whether the object has the resolved dependencies.
- Whether the resolution had errors.
- Whether the downloads were completed.
- Whether the downloads had errors.
- Advice on which methods to call next.

See the example below.

Usage:

```
pkg_download_proposal$print(...)
```

Arguments:

`...` not used currently.

Returns: The `pkg_download_proposal` object itself, invisibly.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
pkg_download_proposal$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Method get_refs()
pdl <- new_pkg_download_proposal(c("pak", "jsonlite"))
pdl$get_refs()


# Method get_config()
pdl <- new_pkg_download_proposal("pak")
pdl$get_config()


# Method resolve()
pdl <- new_pkg_download_proposal("pak")
pdl$resolve()
pdl$get_resolution()


# Method get_resolution()
pdl <- new_pkg_download_proposal("r-lib/pkgdepends")
pdl$resolve()
pdl$get_resolution()


# Method download()
pdl <- new_pkg_download_proposal("r-lib/pkgdepends")
pdl$resolve()
pdl$download()
pdl$get_downloads()


# Method get_downloads()
pdl <- new_pkg_download_proposal("pkgload")
pdl$resolve()
pdl$download()
pdl$get_downloads()


# Method print()
pdl <- new_pkg_download_proposal("r-lib/pkgdepends")
pdl

pdl$resolve()
pdl

pdl$download()
pdl
```

new_pkg_installation_plan

R6 class for installation from a lock file

Description

An installation plan is similar to an installation proposal (i.e. pkg_installation_proposal), but it already contains the solved dependencies, complete with download URLs.

Usage

```
new_pkg_installation_plan(lockfile = "pkg.lock", config = list(), ...)
```

Arguments

lockfile	Path to the lock file to use.
config	Configuration options, a named list. See 'Configuration'. If it does not include library, then .libPaths()[1] is added as library.
...	Additional arguments, passed to pkg_installation_plan$new().

Details

Typically you create a pkg_installation_plan object with new_pkg_installation_plan() and then call its $download() method to download the packages and then its $install() method to install them.

Value

new_pkg_installation_plan() returns a pkg_installation_plan object.

Super class

pkgdepends::pkg_installation_proposal -> pkg_installation_plan

Methods

Public methods:

- pkg_installation_plan$new()
- pkg_installation_plan$resolve()
- pkg_installation_plan$async_resolve()
- pkg_installation_plan$get_solve_policy()
- pkg_installation_plan$set_solve_policy()
- pkg_installation_plan$solve()
- pkg_installation_plan$update()
- pkg_installation_plan$format()
- pkg_installation_plan$clone()

Method `new()`**:** Create a new `pkg_installation_plan` object. Consider using `new_pkg_installation_plan()` instead of calling the constructor directly.

The returned object can be used to download and install packages, according to the plan.

Usage:

```
pkg_installation_plan$new(
  lockfile = "pkg.lock",
  config = list(),
  remote_types = NULL
)
```

Arguments:

`lockfile` Path to the lock file to use.

`config` Configuration options. See 'Configuration'. It needs to include the package library to install to, in `library`.

`remote_types` Custom remote ref types, this is for advanced use, and experimental currently.

Method `resolve()`**:** This function is implemented for installation plans, and will error.

Usage:

```
pkg_installation_plan$resolve()
```

Method `async_resolve()`**:** This function is implemented for installation plans, and will error.

Usage:

```
pkg_installation_plan$async_resolve()
```

Method `get_solve_policy()`**:** Installation plans are already solved, and this method will return `NA_character_`, always.

Usage:

```
pkg_installation_plan$get_solve_policy()
```

Method `set_solve_policy()`**:** This function is implemented for installation plans, and will error.

Usage:

```
pkg_installation_plan$set_solve_policy()
```

Method `solve()`**:** This function is implemented for installation plans, and will error.

Usage:

```
pkg_installation_plan$solve()
```

Method `update()`**:** Update the plan to the current state of the library. If the library has not changed since the plan was created, then it does nothing. If new packages have been installed, then it might not be necessary to download and install all packages in the plan.

Usage:

```
pkg_installation_plan$update()
```

Details: This operation is different than creating a new proposal with the updated library, because it uses the the packages and package versions of the original plan. E.g. if the library has a newer version of a package, then `$update()` will downgrade it to the version in the plan.

Method `format()`**:** Format a `pkg_installation_plan` object, typically for printing.

Usage:

```
pkg_installation_plan$format(...)
```

Arguments:

`...` not used currently.

Returns: A character vector, each element should be a line in the printout.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
pkg_installation_plan$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`new_pkg_installation_proposal` *R6 class for package download and installation.*

Description

Download and install R packages, with their dependencies, from various sources.

Usage

```
new_pkg_installation_proposal(refs, config = list(), ...)
```

Arguments

`refs`	Package names or references. See 'Package references' for the syntax.
`config`	Configuration options, a named list. See 'Configuration'. If it does not include `library`, then `.libPaths()[1]` is added as `library`.
`...`	Additional arguments, passed to `pkg_installation_proposal$new()`.

Details

`new_pkg_installation_proposal()` creates a new object from the `pkg_installation_proposal` class. The advantage of `new_pkg_installation_proposal()` compared to using the pkg_installation_proposal constructor directly is that it avoids making pkgdepends a build time dependency.

Typical workflow to install a set of packages:

1. Create a `pkg_installation_proposal` object with `new_pkg_installation_proposal()`.
2. Resolve all possible dependencies with `pkg_installation_proposal$resolve()`.
3. Solve the package dependencies, to get an installation plan, with `pkg_installation_proposal$solve()`.
4. Download all files with `pkg_installation_proposal$download()`.
5. Install the downloaded files with `pkg_installation_proposal$install()`.

Value

`new_pkg_installation_proposal()` returns a new `pkg_installation_proposal` object.

Methods

Public methods:

- `pkg_installation_proposal$new()`
- `pkg_installation_proposal$get_refs()`
- `pkg_installation_proposal$get_config()`
- `pkg_installation_proposal$resolve()`
- `pkg_installation_proposal$async_resolve()`
- `pkg_installation_proposal$get_resolution()`
- `pkg_installation_proposal$get_solve_policy()`
- `pkg_installation_proposal$set_solve_policy()`
- `pkg_installation_proposal$solve()`
- `pkg_installation_proposal$get_solution()`
- `pkg_installation_proposal$show_solution()`
- `pkg_installation_proposal$stop_for_solution_error()`
- `pkg_installation_proposal$create_lockfile()`
- `pkg_installation_proposal$draw()`
- `pkg_installation_proposal$download()`
- `pkg_installation_proposal$async_download()`
- `pkg_installation_proposal$get_downloads()`
- `pkg_installation_proposal$stop_for_download_error()`
- `pkg_installation_proposal$install()`
- `pkg_installation_proposal$install_sysreqs()`
- `pkg_installation_proposal$get_install_plan()`
- `pkg_installation_proposal$format()`
- `pkg_installation_proposal$print()`
- `pkg_installation_proposal$clone()`

Method `new()`**:** Create a new `pkg_installation_proposal` object. Consider using `new_pkg_installation_proposal(` instead of calling the constructor directly.

The returned object can be used to look up (recursive) dependencies of R packages from various sources, and then download and install the package files.

Usage:

```
pkg_installation_proposal$new(
  refs,
  config = list(),
  policy = c("lazy", "upgrade"),
  remote_types = NULL
)
```

Arguments:

`refs` Package names or references. See 'Package references' for the syntax.

`config` Configuration options, a named list. See 'Configuration'. It needs to include the package library to install to, in `library`.

`policy` Solution policy. See 'The dependency solver'.

`remote_types` Custom remote ref types, this is for advanced use, and experimental currently.

Method `get_refs()`**:** The package refs that were used to create the `pkg_installation_proposal` object.

Usage:

```
pkg_installation_proposal$get_refs()
```

Returns: A character vector of package refs that were used to create the `pkg_installation_proposal` object.

Method `get_config()`**:** Configuration options for the `pkg_installation_proposal` object. See 'Configuration' for details.

Usage:

```
pkg_installation_proposal$get_config()
```

Returns: Named list. See 'Configuration' for the configuration options.

Method `resolve()`**:** Resolve the dependencies of the specified package references. This usually means downloading metadata from CRAN and Bioconductor, unless already cached, and also from GitHub if GitHub refs were included, either directly or indirectly. See 'Dependency resolution' for details.

Usage:

```
pkg_installation_proposal$resolve()
```

Returns: The `pkg_installation_proposal` object, invisibly.

Method `async_resolve()`**:** The same as `resolve()`, but asynchronous. This method is for advanced use.

Usage:

```
pkg_installation_proposal$async_resolve()
```

Returns: A deferred value.

Method `get_resolution()`**:** Query the result of the dependency resolution. This method can be called after `resolve()` has completed.

Usage:

```
pkg_installation_proposal$get_resolution()
```

Returns: A pkg_resolution_result object, which is also a data frame. See 'Dependency resolution' for its columns.

Method `get_solve_policy()`**:** Returns the current policy of the dependency solver. See 'The dependency solver' for details.

Usage:

```
pkg_installation_proposal$get_solve_policy()
```

Returns: A character vector of length one.

Method `set_solve_policy()`**:** Set the current policy of the dependency solver. If the object already contains a solution and the new policy is different than the old policy, then the solution is deleted. See 'The dependency solver' for details.

Usage:

```
pkg_installation_proposal$set_solve_policy(policy = c("lazy", "upgrade"))
```

Arguments:

`policy` Policy to set.

Method `solve()`**:** Solve the package dependencies. Out of the resolved dependencies, it works out a set of packages, that can be installed together to create a functional installation. The set includes all directly specified packages, and all required (or suggested, depending on the configuration) packages as well. It includes every package at most once. See 'The dependency solver' for details.

Usage:

```
pkg_installation_proposal$solve()
```

Returns: The `pkg_installation_proposal` object itself, invisibly.

Method `get_solution()`**:** Returns the solution of the package dependencies.

Usage:

```
pkg_installation_proposal$get_solution()
```

Returns: A pkg_solution_result object, which is a list. See pkg_solution_result for details.

Method `show_solution()`**:** Show the solution on the screen.

Usage:

```
pkg_installation_proposal$show_solution(key = FALSE)
```

Arguments:

`key` Whether to show the key to the package list annotation.

Returns: A pkg_solution_result object, which is a list. See pkg_solution_result for details.

Method `stop_for_solution_error()`**:** Error if the dependency solver failed to find a consistent set of packages that can be installed together.

Usage:

```
pkg_installation_proposal$stop_for_solution_error()
```

Method `create_lockfile()`**:** Create a lock file that contains the information to perform the installation later, possibly in another R session.

Usage:

```
pkg_installation_proposal$create_lockfile(path = "pkg.lock", version = 1)
```

Arguments:

`path` Name of the lock file. The default is `pkg.lock` in the current working directory.

`version` Only version 1 is supported currently.

Details: Note, since the URLs of CRAN and most CRAN-like repositories change over time, in practice you cannot perform the plan of the lock file *much* later. For example, binary packages of older package version are removed, and won't be found.
Similarly, for `url::` remote types, the URL might hold an updated version of the package, compared to when the lock file was created. Should this happen, pkgdepends prints a warning, but it will try to continue the installation. The installation might fail if the updated package has different (e.g. new) dependencies.
Currently the intended use case of lock files in on CI systems, to facilitate caching. The (hash of the) lock file provides a good key for caching systems.

Method `draw()`**:** Draw a tree of package dependencies. It returns a `tree` object, see `cli::tree()`. Printing this object prints the dependency tree to the screen.

Usage:

```
pkg_installation_proposal$draw()
```

Returns: A `tree` object from the cli package, see `cli::tree()`.

Method `download()`**:** Download all packages that are part of the solution. It uses the package cache in the pkgcache package by default, to avoid downloads if possible.

Usage:

```
pkg_installation_proposal$download()
```

Returns: The `pkg_installation_proposal` object itself, invisibly.

Method `async_download()`**:** The same as `download()`, but asynchronous. This method is for advanced use.

Usage:

```
pkg_installation_proposal$async_download()
```

Returns: A deferred value.

Method `get_downloads()`**:** Returns the summary of the package downloads.

Usage:

```
pkg_installation_proposal$get_downloads()
```

Returns: A pkg_download_result object, which is a list. See pkg_download_result for details.

Method `stop_for_download_error()`**:** Throw and error if the some of the downloads have failed for the most recent `pkg_installation_proposal$download()` call.

Usage:

```
pkg_installation_proposal$stop_for_download_error()
```

Method `install()`**:** Install the downloaded packages. It calls `install_package_plan()`.

Usage:

```
pkg_installation_proposal$install()
```

Returns: The return value of `install_package_plan()`.

Method `install_sysreqs()`**:** Install system requirements. It does nothing if system requirements are turned off. It errors if we could not look up the system requirements. Create an installation plan for the downloaded packages.

Usage:

```
pkg_installation_proposal$install_sysreqs()
```

Method `get_install_plan()`**:**

Usage:

```
pkg_installation_proposal$get_install_plan()
```

Returns: An installation plan, see 'Installation plans' for the format.

Method `format()`**:** Format a `pkg_installation_proposal` object, typically for printing.

Usage:

```
pkg_installation_proposal$format(...)
```

Arguments:

`...` not used currently.

Returns: A character vector, each element should be a line in the printout.

Method `print()`**:** Prints a `pkg_installation_proposal` object to the screen.
The printout includes:

- The package refs.
- The policy of the dependency solver.
- Whether the object has the solved dependencies.
- Whether the solution had errors.
- Whether the object has downloads.
- Whether the downloads had errors.
- Advice on which methods to call next.

See the example below.

Usage:

```
pkg_installation_proposal$print(...)
```

Arguments:

`...` not used currently.

Returns: The `pkg_installation_proposal` object itself, invisibly.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
pkg_installation_proposal$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
pdi <- new_pkg_installation_proposal(
  "pak",
  config = list(library = tempfile())
)
pdi

pdi$resolve()
pdi

pdi$solve()
pdi

pdi$download()
pdi

## End(Not run)


pdi <- new_pkg_installation_proposal(
  "r-lib/pkgdepends",
  config = list(library = tempfile()))
pdi


pdi <- new_pkg_installation_proposal("r-lib/pkgdepends")
pdi$get_refs()


pdi <- new_pkg_installation_proposal(
  "pak",
  config = list(library = tempfile())
)
pdi$get_config()


## Not run:
pdi <- new_pkg_installation_proposal(
  "pak",
  config = list(library = tempfile())
)

pdi$resolve()
pdi$get_resolution()

## End(Not run)


## Not run:
pdi <- new_pkg_installation_proposal(
```

```
  "r-lib/pkgdepends",
  config = list(library = tempfile())
)
pdi$resolve()
pdi$get_resolution()

## End(Not run)


pdi <- new_pkg_installation_proposal(
  "r-lib/pkgdepends",
  config = list(library = tempfile())
)
pdi$get_solve_policy()
pdi$set_solve_policy("upgrade")
pdi$get_solve_policy()


pdi <- new_pkg_installation_proposal(
  "r-lib/pkgdepends",
  config = list(library = tempfile())
)
pdi$get_solve_policy()
pdi$set_solve_policy("upgrade")
pdi$get_solve_policy()


## Not run:
pdi <- new_pkg_installation_proposal(
  "r-lib/pkgdepends",
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$get_solution()

## End(Not run)


## Not run:
pdi <- new_pkg_installation_proposal(
  "r-lib/pkgdepends",
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$get_solution()

## End(Not run)


## Not run:
pdi <- new_pkg_installation_proposal(
```

```
  "r-lib/pkgdepends",
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$get_solution()
pdi$show_solution()

## End(Not run)


## Not run:
# This is an error, because the packages conflict:
pdi <- new_pkg_installation_proposal(
  c("r-lib/pak", "cran::pak"),
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi
# This fails:
# pdi$stop_for_solution_error()

## End(Not run)


## Not run:
pdi <- new_pkg_installation_proposal(
  "pak",
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$draw()

## End(Not run)


## Not run:
pdi <- new_pkg_installation_proposal(
  c("r-lib/pak", "cran::pak"),
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$download()
pdi$get_downloads()

## End(Not run)


## Not run:
pdi <- new_pkg_installation_proposal(
```

```
  c("r-lib/pak", "cran::pak"),
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$download()
pdi$get_downloads()

## End(Not run)


## Not run:
pdi <- new_pkg_installation_proposal(
  "pak",
  config = list(library = tempfile())
)
pdi$resolve()
pdi$solve()
pdi$download()
pdi$get_install_plan()

## End(Not run)


# Method print
pdi <- new_pkg_installation_proposal(
  "pak",
  config = list(library = tempfile())
)
pdi

pdi$resolve()
pdi

pdi$solve()
pdi

pdi$download()
pdi
```

`parse_pkg_refs` *Parse package location references*

Description

See pkg_refs for more about supported package references.

Usage

```
parse_pkg_refs(refs, remote_types = NULL, ...)

parse_pkg_ref(ref, remote_types = NULL, ...)
```

Arguments

refs	Character vector of references.
remote_types	Custom remote types can be added here, this is for advanced use, and experimental currently.
...	Additional arguments are passed to the individual parser functions.
ref	A package reference, like refs, but a length one vector, for convenience.

Value

`parse_pkg_refs()` returns a list of parsed references. `parse_pkg_ref()` returns one parsed reference. A parsed reference is a list, with at least elements:

- `ref`: The original reference string.
- `type`: The reference type.
- `package`: The package name. It typically contains additional data, specific to the various reference types. See pkg_refs for details. The parsed reference always has class `remote_ref_<type>` and `remote_ref`.

---

pkg_config *pkgdepends configuration*

Description

Configuration entries for several pkgdepends classes.

Usage

```
current_config()
```

Details

pkgdepends configuration is set from several source. They are, in the order of preference:

- Function arguments, e.g. the `config` argument of `new_pkg_installation_proposal()`.
- Global options, set via `options()`. The name of the global option is the `pkg.` prefix plus the name of the pkgdepends configuration entry. E.g. `pkg.platforms`.
- Environment variables. The name of the environment variable is the `PKG_` prefix, plus the name of the pkgdepends configuration entry, in uppercase. E.g. `PKG_PLATFORMS`.
- Default values.

Not all classes use all entries. E.g. a pkg_download_proposal is not concerned about package libraries, so it'll ignore the library configuration entry.

Call current_config() to print the current configuration.

Configuration entries

- build_vignettes: Whether to build vignettes for package trees. This is only used if the package is obtained from a package tree, and not from a source (or binary) package archive. By default vignettes are not built in this case. If you set this to TRUE, then you need to make sure that the vignette builder packages are available, as these are not installed by default currently.
- cache_dir: Directory to download the packages to. Defaults to a temporary directory within the R session temporary directory, see base::tempdir().
- cran_mirror: CRAN mirror to use. Defaults to the repos option (see base::options()), if that's not set then https://cran.rstudio.com.
- dependencies: Dependencies to consider or download or install. Defaults to the hard dependencies, see pkg_dep_types_hard(). The following values are supported in the PKG_DEPENDENCIES environment variable: "TRUE", "FALSE", "NA", or a semicolon separated list of dependency types. See as_pkg_dependencies() for details.
- library: Package library to install packages to. It is also used for already installed packages when considering dependencies in dependency lookup or package installation. Defaults to the first path in .libPaths().
- metadata_cache_dir: Location of metadata replica of pkgcache::cranlike_metadata_cache. Defaults to a temporary directory within the R session temporary directory, see base::tempdir().
- metadata_update_after: A time interval as a difftime object. pkgdepends will update the metadata cache if it is older than this. The default is one day. The PKG_METADATA_UPDATE_AFTER environment variable may be set in seconds (s suffix), minutes (m suffix), hours (h suffix), or days (d suffix). E.g: 1d means one day.
- package_cache_dir: Package cache location of pkgcache::package_cache. The default is the pkgcache default.
- platforms: Character vector of platforms to *download* or *install* packages for. See default_platforms() for possible platform names. Defaults to the platform of the current R session, plus "source".
- r_versions: Character vector, R versions to download or install packages for. It defaults to the current R version.
- sysreqs: Whether to look up and install system requirements. By default this is TRUE if the CI environment variable is set and the operating system is a supported Linux distribution: CentOS, openSUSE, RedHat Linux, Ubuntu Linux or SUSE Linux Enterprise. The default will change as new platforms gain system requirements support.
- sysreqs_dry_run: If TRUE, then pkgdepends only prints the system commands to install system requirements, but does not execute them.
- sysreqs_rspm_repo_id: Posit Package Manager (formerly RStudio Package Manager) repository id to use for CRAN system requirements lookup. Defaults to the RSPM_REPO_ID environment variable, if set. If not set, then it defaults to 1.
- sysreqs_rspm_url: Root URL of Posit Package Manager (formerly RStudio Package Manager) for system requirements lookup. By default the RSPM_ROOT environment variable is used, if set. If not set, it defaults to https://packagemanager.posit.co.

- `sysreqs_sudo`: Whether to use `sudo` to install system requirements, on Unix. By default it is `TRUE` on Linux if the effective user id of the current process is not the `root` user.
- `sysreqs_verbose`: Whether to echo the output of system requirements installation. Defaults to `TRUE` if the `CI` environment variable is set.
- `use_bioconductor`: Whether to automatically use the Bioconductor repositories. Defaults to `TRUE`.
- `windows_archs`: Character scalar specifying which architectures to download/install for on Windows. Its possible values are:
  - `"prefer-x64"`: Generally prefer x64 binaries. If the current R session is `x64`, then we download/install x64 packages. (These packages might still be multi-architecture binaries!) If the current R session is `i386`, then we download/install packages for both architectures. This might mean compiling packages from source if the binary packages are for `x64` only, like the CRAN Windows binaries for R 4.2.x currently. `"prefer-x64"` is the default for R 4.2.0 and later.
  - `"both"`: Always download/install packages for both `i386` and `x64` architectures. This might need compilation from source if the available binaries are for `x64` only, like the CRAN Windows binaries for R 4.2.x currently. `"both"` is the default for R 4.2.0 and earlier.

`pkg_dep_types_hard` *Possible package dependency types*

Description

Hard dependencies are needed for a package to load, soft dependencies are optional.

Usage

```
pkg_dep_types_hard()

pkg_dep_types_soft()

pkg_dep_types()
```

Value

A string vector of dependency types, capitalized.

See Also

Other package dependency utilities: `as_pkg_dependencies()`

pkg_downloads *Package downloads*

Description

The pkg_download_proposal and pkg_installation_proposal classes both have download methods, to downloads package files into a configured directory (see 'Configuration').

Details

They return a pkg_download_result object, which is a data frame, that adds extra columns to pkg_resolution_result (for pkg_download_proposal) or pkg_solution_result (for pkg_installation_proposal):

- built: the Built field from the DESCRIPTION file of binary packages, for which this information is available.
- cache_status: whether the package file is in the package cache. It is NA for installed:: package refs.
- dep_types: character vector of dependency types that were considered for this package. (This is a list column.)
- deps: dependencies of the package, in a data frame. See "Package dependency tables" below.
- direct: whether this package (ref, really) was directly specified, or added as a dependency.
- error: this is a list column that contains error objects for the refs that pkgdepends failed to resolve.
- filesize: the file size in bytes, or NA if this information is not available.
- license: license of the package, or NA if not available.
- md5sum: MD5 checksum of the package file, if available, or NA if not.
- metadata: a named character vector. These fields will be (should be) added to the installed DESCRIPTION file of the package.
- mirror: URL of the CRAN(-like) mirror site where the metadata was obtained from. It is NA for non-CRAN-like sources, e.g. local files, installed packages, GitHub, etc.
- needscompilation: whether the package needs compilation.
- package: package name.
- priority: this is "base" for base packages, "recommended" for recommended packages, and NA otherwise.
- ref: package reference.
- remote: the parsed remote_ref objects, see parse_pkg_refs(). This is a list column.
- repodir: the directory where this package should be in a CRAN-like repository.
- sha256: SHA256 hash of the package file, if available, otherwise NA.
- sources: URLs where this package can be downloaded from. This is a zero length vector for installed:: refs.
- status: status of the dependency resolution, "OK" or "FAILED".

- `target`: path where this package should be saved in a CRAN-repository.
- `type`: ref type.
- `version`: package version.
- `fulltarget`: absolute path to the downloaded file. At most one of `fulltarget` and `fulltarget_tree` must exist on the disk.
- `fulltarget_tree`: absolute path to a package tree directory. At most one of `fulltarget` and `fulltarget_tree` must exist on the disk.
- `download_status`: `"Had"` or `"Got"`, depending on whether the file was obtained from the cache.
- `download_error`: error object for failed downloads.
- `file_size`: Size of the file, or `NA`. For `installed::` refs, it is `NA`, and it is also `NA` for refs that created `fulltarget_tree` instead of `fulltarget`.

`fulltarget`, if it exists, contains a packaged (via `R CMD build`) source R package. If `fulltarget_tree` exists, it is a package tree directory, that still needs an `R CMD build` call.

Additional columns might be present. They are either used internally or they are experimental. They might be removed or changed at any time.

All columns are of type character, except for `direct` (logical), `needscompilation` (logical), `filesize` (integer), `deps` (list column, see "Package dependency tables" below), `sources` (list of character vectors), `remote` (list), `error` (list), `metadata` (list), `dep_types` (list).

Package dependency tables:

A package dependency tables in the `deps` list column have five columns currently:

- `ref`: the package ref of the dependency.
- `type`: the dependency type, in all lowercase. I.e. `imports`, `suggests`, etc.
- `package`: package name of the dependency.
- `op`: operator for version requirements, e.g. `>=`.
- `version`: version number, for version requirements.

`pkg_name_check` *Check if an R package name is available.*

Description

Additionally, look up the candidate name in a number of dictionaries, to make sure that it does not have a negative meaning.

Usage

```
pkg_name_check(name, dictionaries = NULL)
```

Arguments

name	Package name candidate.
dictionaries	Character vector, the dictionaries to query. Available dictionaries: * wikipedia * wiktionary, * acromine (http://www.nactem.ac.uk/software/acromine/), * sentiment (https://github.com/fnielsen/afinn), * urban (Urban Dictionary). If NULL (by default), the Urban Dictionary is omitted, as it is often offensive.

Details

Valid package name check:

Check the validity of name as a package name. See 'Writing R Extensions' for the allowed package names. Also checked against a list of names that are known to cause problems.

CRAN checks:

Check name against the names of all past and current packages on CRAN, including base and recommended packages.

Bioconductor checks:

Check name against all past and current Bioconductor packages.

Profanity check:

Check name with https://www.purgomalum.com/service/containsprofanity to make sure it is not a profanity.

Dictionaries:

See the dictionaries argument.

Value

pkg_name_check object with a custom print method.

Examples

```
pkg_name_check("cli")
```

---

pkg_refs *Package references*

Description

A package reference (ref) specifies a location from which an R package can be obtained from. The full syntax of a reference is type::ref, but type can be often omitted, the common ref types have shortcuts.

Package references

Many pkgdepends and pak functions take package names as arguments. E.g. `pak::pkg_install()` takes the names of the packages to install, `pak::pkg_deps_tree()` takes the names of the packages to draw dependency trees for.

Most of these function can also take a more generic *package reference* instead of a package name. A package reference also tells pkgdepends where to find the package, the package source.

To specify a package source, use its name as a prefix, with a `::` separator. E.g. `cran::mypkg` means the `mypkg` package from CRAN.

A package name is a special package reference, that implicitly specifies the configured CRAN(-like) repositories as the package source. (We call this the `standard` package source.) So `mypkg` is equivalent to `standard::mypkg` and pak look for mypkg in any of the configured CRAN-like repositories. If you did not explicitly specify any CRAN-like repositories (e.g. with `options("repos")`), then pak uses the CRAN and Bioconductor repositories by default.

This is the list of the currently supported package sources. We will discuss each in detail below.

- `cran`: a CRAN package.
- `bioc`: a Bioconductor package.
- `standard`: a package from a configured CRAN-like repository.
- `github`: a package from GitHub.
- `local`: a local package file or directory.
- `url`: an URL that points to a package archive.
- `installed` an installed package.
- `deps` the dependencies of a local package file or directory.
- `any` a special reference type that accepts a package from any source. See below.
- `param` a special reference to change how other references are downloaded or installed. See "Parameters" below.

Shorthands:

To save typing, you do not always need to fully specify the package source in a package reference. You have seen before that a package name implicitly has a `standard` package source. Here are the complete rules for such shorthands, in the order they are applied:

- If the ref is a valid package name, or a package name with a `@` version specification, the `standard` package source is used. E.g. `pkg` is equivalent to `standard::pkg` and `pkg@1.0` is equivalent to `standard::pkg@1.0`.
- If the ref is a valid `github` ref type without the `github::` prefix, then `github` is used. E.g. `user/repo` is equivalent to `github::user/repo` and `user/repo@tag` is equivalent to `github::user/repo@tag`, etc.
- If the ref is a GitHub URL (see below) without the `github::` prefix, then `github` is used.
- If the ref is a path that starts with `.` or `/` or `\` or `~` then `local` is used. (pkgdepends does not check if the path exists.)
- If a package reference if of the form `<package-name>=?<parameters>`, then it will be the special `param` type. See "Parameters" below.

If the package reference does not have an explicit package source, and the package source cannot be determined from these rules, then pkgdepends throws an error.

Package names:

When pkgdepends is looking up the dependencies of a package, it needs to be able to determine the name of the dependency from the package reference. This is sometimes not easy for dependencies in `Remotes` (or similar) fields.

- For `github::` dependencies pkgdepends assumes that the package name is the same as the name of the repository. If this does not hold, then you need to specify the package name explicitly, using a `<package>=` prefix. E.g. `pins=rstudio/pins-r`. If you specify both the package source type and the package name, the package name comes first: `pins=github::rstudio/pins-r`.
- For `local::` dependencies, you always need to specify the package name explicitly. E.g. `pins=local::~/works/pins`.
- For `url::` dependencies, you always need to specify the package name explicitly. E.g. `ggplot2=url::https://cloud.r-project.org/src/contrib/....`

Parameters:

Package references may have optional parameters, added after a question mark. Different parameters are separated by an ampersand (`&`) character. (This is very similar to how HTTP URLs take query parameters.)

Parameters may be flags that turn on some behavior, or they can have a string value, assigned with an equal sign (`=`). If no value is assigned, then we assume the `true` value. For example these two package refs are equivalent:

```
cran::testthat?source&nocache
cran::testthat?source=true&nocache=true
```

Parameters for downstream packages:

pkgdepends allows specifying parameters for downstream packages, using the `<package>=?<params>` special package reference, where `package` is the name of the package, and `<params>` are the parameters, as above. This is useful if you want to add a parameter to a downstream dependency. For example, to install ggplot2, and always reinstall its cli package dependency you could use the `ggplot2` and `cli=?reinstall` package references. The latter tells pkgdepends to always reinstall cli, even if it is already installed.

Currently supported parameters:

- `ignore` is a flag parameter. If specified, the package is ignored. This usually makes sense in the `packagename=?ignore` form, to ignore a downstream soft dependency. If all versions of a hard dependency are ignored that will lead to a solution error.
- `ignore-before-r` is a version number parameter. The package will be ignored on R versions that are older than the specified one. E.g. `Matrix=?ignore-before-r=4.1.2` will ignore the Matrix package on R versions that are older than 4.1.2. This parameter really only makes sense in the `packgename=?ignore` form.
- `source` is a flag parameter. If specified, then a source R package is requested from a CRAN-like repository. For package installations `source` always triggers a re-install. In other words, `source` implies the `reinstall` parameter. This parameter is supported for `bioc::`, `cran::` and `standard::` remote types, and it is ignored for others.
- `reinstall` requests a re-install for package installations. It is supported by the `bioc::`, `cran::`, `github::`, `local::`, `standard::`, and `url::` remote types.

- nocache will ignore the package cache. It will always download the package file, and it will not add the downloaded (and built) package(s) to the package cache. It is supported by the bioc::, cran::, github::, standard:: and url:: remote types.

Package source details:

*CRAN packages (*cran::*):*
A package from CRAN. Full syntax:

[cran::]<package>[@[>=]<version> | @current | @last]

- <package> is a valid package name.
- <version> is a version or a version requirement.

Examples:

```
forecast
forecast@8.8
forecast@>=8.8
cran::forecast
forecast@last
forecast@current
```

Note: pkgdepends currently parses the version specification part (everything after @), but does not use it.

*Bioconductor packages (*bioc::*):*
A package from Bioconductor. The syntax is the same as for CRAN packages, except for the prefix.

[bioc::]<package>[@[>=]<version> | @current | @last]

*Standard packages (*standard::*):*
These are packages either from CRAN or Bioconductor, the full syntax is the same as for CRAN packages, except for the prefix:

[standard::]<package>[@[>=]<version> | current | last]

*GitHub packages (*github::*):*
Packages from a GitHub repository. Full syntax:

[<package>=][github::]<username>/<repository>[/<subdir>][<detail>]

- <package> is the name of the package. If this is missing, then the name of the repository is used.
- <username> is a GitHub username or organization name.
- <repository> is the name of the repository.
- <subdir> optional subdirectory, if the package is within a subdirectory in the repository.
- <detail> specifies a certain version of the package, see below.

<detail> may specify:

- a git branch, tag or (prefix of) a commit hash: @<commitish>;
- a pull request: #<pull-request>; or
- the latest release: @*release.

If <detail> is missing, then the latest commit of the *default* branch is used.
Examples:

```
r-lib/crayon
github::r-lib/crayon
r-lib/crayon@84be6207
r-lib/crayon@branch
r-lib/crayon#41
r-lib/crayon@release
```

For convenience GitHub HTTP URLs can also be used to specify a package from GitHub. Examples:

```
https://github.com/r-lib/withr
# A branch:
https://github.com/r-lib/withr/tree/ghactions
# A tag:
https://github.com/r-lib/withr/tree/v2.1.1
# A commit:
https://github.com/r-lib/withr/commit/8fbcb548e316
# A pull request:
https://github.com/r-lib/withr/pull/76
# A release:
https://github.com/r-lib/withr/releases/tag/v2.1.0
```

A GitHub remote string can also be used instead of an URL, for example: `git@github.com:r-lib/pak.git`

Local packages (`local::`):
A path that refers to a package file built with `R CMD build`, or a directory that contains a package. Full syntax:

```
local::<path>
```

For brevity, you can omit the `local::` prefix, if you specify an absolute path, a path from the user's home directory, starting with `~`, or a relative path starting with `./` or `.\\`.
A single dot (`"."`) is considered to be a local package in the current working directory.
Examples:

```
local::/foo/bar/package_1.0.0.tar.gz
local::/foo/bar/pkg
local::.
/absolute/path/package_1.0.0.tar.gz
~/path/from/home
./relative/path
.
```

If you specify a local package in a dependency (i.e. in `DESCRIPTION`), then you also need to specify the name of the package, see "Package names" above.

URLs (`url::`):
You can use `url::` to refer to URLs that hold R package archives (i.e. properly built with `R CMD build`), or compressed directories of package trees (i.e. not built with `R CMD build`). pkgdepends will figure out if it needs to run `R CMD build` on the package first.
This remote type supports `.tar.gz` and `.zip` files.
Note that URLs are not ideal remote types, because pkgdepends needs to download the package file to resolve its dependencies. When this happens, it puts the package file in the cache, so no further downloads are needed when installing the package later.
Examples:

```
url::https://cloud.r-project.org/src/contrib/Archive/cli/cli_1.0.0.tar.gz
url::https://github.com/tidyverse/stringr/archive/HEAD.zip
```

If you specify a package from an URL in a dependency (i.e. in `DESCRIPTION`), then you also need to specify the name of the package, see "Package names" above.

Installed packages (`installed::`):
This is usually used internally, but can also be used directly. Full syntax:

`installed::<path>/<package>`

- `<path>` is the library the package is installed to.
- `<package>` is the package name.

Example:

```
installed::~/R/3.6/crayon
```

Package dependencies (`deps::`):
Usually used internally, it specifies the dependencies of a local package. It can be used to download or install the dependencies of a package, without downloading or installing the package itself. Full syntax:

`deps::<path>`

Examples:

```
deps::/foo/bar/package_1.0.0.tar.gz
deps::/foo/bar/pkg
deps::.
```

`any::` *packages:*
Sometimes you need to install additional packages, but you don't mind where they are installed from. Here is an example. You want to install cli from GitHub, from `r-lib/cli`. You also want to install glue, and you don't mind which version of glue is installed, as long as it is compatible with the requested cli version. If cli specifies the development version of glue, then that is fine. If cli is fine with the CRAN version of glue, that's OK, too. If a future version of cli does not depend on glue, you still want glue installed, from CRAN. The `any::` reference type does exactly this.
In our example you might write

```
pak::pkg_install(c("glue", "r-lib/cli"))
```

first, but this will fail if `rlib/cli` requests (say) `tidyverse/glue`, because in `pkg_install()` `"glue"` is interpreted as `"standard::glue"`, creating a conflict with `tidyverse/glue`. On the other hand

```
pak::pkg_install(c("any::glue", "r-lib/cli"))
```

works, independently of which glue version is requested by cli.

Parameter refs (`param::`):
See "Parameters" above.

The `Remotes` field:

In the `DESCRIPTION` file of an R package you can mark any regular dependency defined in the `Depends`, `Imports`, `Suggests` or `Enhances` fields as being installed from a non-standard package source by adding a package reference to a `Remotes` entry. pkgdepends will download and install the package from the from the specified location, instead of a CRAN-like repository.

The remote dependencies specified in `Remotes` is a comma separated list of package sources:

```
Remotes: <pkg-source-1>, <pkg-source-2>, [ ... ]
```

Note that you will still need add the package to one of the regular dependency fields, i.e. `Imports`, `Suggests`, etc. Here is a concrete example that specifies the `r-lib/glue` package:

```
Imports: glue
Remotes: r-lib/glue,
  r-lib/httr@v0.4,
  klutometis/roxygen#142,
  r-lib/testthat@c67018fa4970
```

The CRAN and Bioconductor repositories do not support the `Remotes` field, so you need to remove this field, before submitting your package to either of them.

`pkg_resolution` *Dependency resolution*

Description

Collect information about dependencies of R packages, recursively.

Details

`pkg_deps`, `pkg_download_proposal` and `pkg_installation_proposal` all resolve their dependencies recursively, to obtain information about all packages needed for the specified package references.

CRAN and Bioconductor packages:
Resolution currently start by downloading the CRAN and Bioconductor metadata, if it is out of date. For CRAN, we also download additional metadata, that includes file sizes, SHA hashes, system requirements, and "built" (for binary packages) and "packaged" time stamps. The extra meta information is updated daily currently, so for some packages it might be incorrect or missing.

GitHub packages:
For GitHub packages, we query their download URL to be able to download the package later, and also download their `DESCRIPTION` file, to learn about their dependencies.

Local packages:
From local package files we extract the `DESCRIPTION` file, to learn about their dependencies.

The `remotes` field in `DESCRIPTION`:
We support the non-standard `Remotes` field in the package `DESCRIPTION` file. This field may contain a list of package references for any of the dependencies that are specified in one of the `Depends`, `Includes`, `Suggests` or `Enhances` fields. The syntax is a comma separated list of package references.

The result:
The result of the resolution is a data frame with information about the packages and their dependencies.

- `built`: the `Built` field from the `DESCRIPTION` file of binary packages, for which this information is available.
- `cache_status`: whether the package file is in the package cache. It is `NA` for `installed::` package refs.
- `dep_types`: character vector of dependency types that were considered for this package. (This is a list column.)
- `deps`: dependencies of the package, in a data frame. See "Package dependency tables" below.
- `direct`: whether this package (ref, really) was directly specified, or added as a dependency.
- `error`: this is a list column that contains error objects for the refs that pkgdepends failed to resolve.
- `filesize`: the file size in bytes, or `NA` if this information is not available.
- `license`: license of the package, or `NA` if not available.
- `md5sum`: MD5 checksum of the package file, if available, or `NA` if not.
- `metadata`: a named character vector. These fields will be (should be) added to the installed `DESCRIPTION` file of the package.
- `mirror`: URL of the CRAN(-like) mirror site where the metadata was obtained from. It is NA for non-CRAN-like sources, e.g. local files, installed packages, GitHub, etc.
- `needscompilation`: whether the package needs compilation.
- `package`: package name.
- `priority`: this is `"base"` for base packages, `"recommended"` for recommended packages, and `NA` otherwise.
- `ref`: package reference.
- `remote`: the parsed `remote_ref` objects, see `parse_pkg_refs()`. This is a list column.
- `repodir`: the directory where this package should be in a CRAN-like repository.
- `sha256`: SHA256 hash of the package file, if available, otherwise `NA`.
- `sources`: URLs where this package can be downloaded from. This is a zero length vector for `installed::` refs.
- `status`: status of the dependency resolution, `"OK"` or `"FAILED"`.
- `target`: path where this package should be saved in a CRAN-repository.
- `type`: ref type.
- `version`: package version.

Additional columns might be present. They are either used internally or they are experimental. They might be removed or changed at any time.

All columns are of type character, except for `direct` (logical), `needscompilation` (logical), `filesize` (integer), `deps` (list column, see "Package dependency tables" below), `sources` (list of character vectors), `remote` (list), `error` (list), `metadata` (list), `dep_types` (list).

Package dependency tables:

A package dependency tables in the `deps` list column have five columns currently:

- `ref`: the package ref of the dependency.
- `type`: the dependency type, in all lowercase. I.e. `imports`, `suggests`, etc.
- `package`: package name of the dependency.
- `op`: operator for version requirements, e.g. `>=`.
- `version`: version number, for version requirements.

Resolution failures:

The resolution process does not stop on error. Instead, failed resolutions return and error object in the `error` column of the result data frame.

`pkg_rx` *A set of handy regular expressions related to R packages*

Description

If you use these in R, make sure you specify `perl = TRUE`, see `base::grep()`.

Usage

```
pkg_rx()
```

Details

Currently included:

- `pkg_name`: A valid package name.
- `type_cran`: A `cran::` package reference.
- `type_bioc`: A `bioc::` package reference.
- `type_standard`: A `standard::` package reference.
- `type_github`: A `github::` package reference.
- `type_local`: A `local::` package reference.
- `type_deps`: A `deps::` package reference.
- `type_installed`: An `installed::` package reference.
- `github_username`: A GitHub username.
- `github_repo`: A GitHub repository name.
- `github_url`: A GitHub URL.

Value

A named list of strings.

Examples

```
pkg_rx()
```

`pkg_solution` *The dependency solver*

Description

The dependency solver takes the resolution information, and works out the exact versions of each package that must be installed, such that version and other requirements are satisfied.

Details

Solution policies:

The dependency solver currently supports two policies: `lazy` and `upgrade`. The `lazy` policy prefers to minimize installation time, and it does not perform package upgrades, unless version requirements require them. The `upgrade` policy prefers to update all package to their latest possible versions, but it still considers that version requirements.

The integer problem:

Solving the package dependencies requires solving an integer linear problem (ILP). This subsection briefly describes how the problem is represented as an integer problem, and what the solution policies exactly mean.

Every row of the package resolution is a candidate for the dependency solver. In the integer problem, every candidate corresponds to a binary variable. This is 1 if that candidate is selected as part of the solution, and 0 otherwise.

The objective of the ILP minimization is defined differently for different solution policies. The ILP conditions are the same.

1. For the `lazy` policy, `installed::` packaged get 0 points, binary packages 1 point, sources packages 5 points.
2. For the 'upgrade' policy, we rank all candidates for a given package according to their version numbers, and assign more points to older versions. Points are assigned by 100 and candidates with equal versions get equal points. We still prefer installed packages to binaries to source packages, so also add 0 point for already installed candidates, 1 extra points for binaries and 5 points for source packages.
3. For directly specified refs, we aim to install each package exactly once. So for these we require that the variables corresponding to the same package sum up to 1.
4. For non-direct refs (i.e. dependencies), we require that the variables corresponding to the same package sum up to at most one. Since every candidate has at least 1 point in the objective function of the minimization problem, non-needed dependencies will be omitted.
5. For direct refs, we require that their candidates satisfy their references. What this means exactly depends on the ref types. E.g. for CRAN packages, it means that a CRAN candidate must be selected. For a standard ref, a GitHub candidate is OK as well.
6. We rule out candidates for which the dependency resolution failed.
7. We go over all the dependency requirements and rule out packages that do not meet them. For every package `A`, that requires package `B`, we select the `B(i, i=1..k)` candidates of `B` that satisfy `A`'s requirements and add a `A - B(1) - ... - B(k) <= 0` rule. To satisfy this rule, either we cannot install `A`, or if `A` is installed, then one of the good `B` candidates must be installed as well.

8. We rule out non-installed CRAN and Bioconductor candidates for packages that have an already installed candidate with the same exact version.
9. We also rule out source CRAN and Bioconductor candidates for packages that have a binary candidate with the same exact version.

Explaining why the solver failed:

To be able to explain why a solution attempt failed, we also add a dummy variable for each directly required package. This dummy variable has a very large objective value, and it is only selected if there is no way to install the directly required package.

After a failed solution, we look the dummy variables that were selected, to see which directly required package failed to solve. Then we check which rule(s) ruled out the installation of these packages, and their dependencies, recursively.

The result:

The result of the solution is a `pkg_solution_result` object. It is a named list with entries:

- `status`: Status of the solution attempt, `"OK"` or `"FAILED"`.
- `data`: The selected candidates. This is very similar to a pkg_resolution_result object, but it has two extra columns:
  - `lib_status`: status of the package in the library, after the installation. Possible values: `new` (will be newly installed), `current` (up to date, not installed), `update` (will be updated), `no-update` (could update, but will not).
  - `old_version`: The old (current) version of the package in the library, or `NA` if the package is currently not installed.
- `problem`: The ILP problem. The exact representation is an implementation detail, but it does have an informative print method.
- `solution`: The return value of the internal solver.

Index

∗ **package dependency utilities**
    as_pkg_dependencies, 5
    pkg_dep_types_hard, 36
∗ **platform functions**
    current_r_platform, 6
'Configuration', *7*, *13*, *18, 19*, *22–24*, *26*, *37*
'Dependency resolution', *13, 14*, *19*, *26*
'Installation plans', *8*, *29*
'Package references', *12, 13*, *17, 18*, *24*, *26*
'The dependency solver', *13, 14*, *26, 27*
.libPaths(), *35*
"Configuration", *4*
"Dependency resolution", *4*
"Installation plans", *4*
"Package references", *3*
"The dependency solver", *4*

as_pkg_dependencies, 5, *36*
as_pkg_dependencies(), *35*

base::grep(), *47*
base::options(), *35*
base::tempdir(), *35*

cli::tree(), *15*, *28*
current_config *(*pkg_config*)*, 34
current_r_platform, 6

default_platforms *(*current_r_platform*)*, 6
default_platforms(), *35*
dependency lookup, *35*
difftime, *35*
downloading, *8*

install_package_plan, 8
install_package_plan(), *9*, *28*
install_plans, 8
installation proposal, *8*
is_valid_package_name, 10

lib_status, 10

new_pkg_deps, 12
new_pkg_download_proposal, 17
new_pkg_installation_plan, 21
new_pkg_installation_proposal, 24
new_pkg_installation_proposal(), *34*

options(), *34*

package installation, *35*
package reference, *9*
package references, *4*, *45*
parse_pkg_ref *(*parse_pkg_refs*)*, 33
parse_pkg_refs, 33
parse_pkg_refs(), *37*, *46*
pkg_config, 34
pkg_dep_types *(*pkg_dep_types_hard*)*, 36
pkg_dep_types_hard, *6*, 36
pkg_dep_types_hard(), *6*, *35*
pkg_dep_types_soft *(*pkg_dep_types_hard*)*, 36
pkg_deps, *3, 4*, *12*, *45*
pkg_deps *(*new_pkg_deps*)*, 12
pkg_download_proposal, *3, 4*, *18*, *35*, *37*, *45*
pkg_download_proposal *(*new_pkg_download_proposal*)*, 17
pkg_download_result, *9*, *20*, *28*
pkg_download_result *(*pkg_downloads*)*, 37
pkg_downloads, 37
pkg_installation_plan *(*new_pkg_installation_plan*)*, 22
pkg_installation_proposal, *4*, *9*, *22*, *24*, *37*, *45*
pkg_installation_proposal *(*new_pkg_installation_proposal*)*, 24
pkg_name_check, 38
pkg_refs, *33, 34*, 39
pkg_resolution, 45

pkg_resolution_result, *14*, *19*, *26*, *37*, *49*
pkg_resolution_result *(*pkg_resolution*)*, 45
pkg_rx, 47
pkg_solution, 48
pkg_solution_result, *14*, *27*, *37*
pkg_solution_result *(*pkg_solution*)*, 48
pkgcache::cranlike_metadata_cache, *35*
pkgcache::package_cache, *35*
pkgdepends *(*pkgdepends-package*)*, 2
pkgdepends-config *(*pkg_config*)*, 34
pkgdepends-package, 2
pkgdepends::pkg_installation_proposal, *22*

solving, *8*

utils::install.packages(), *5*