Package `pak`

October 14, 2022

Version 0.3.1
Title Another Approach to Package Installation
Description The goal of `pak` is to make package installation faster and more reliable. In particular, it performs all HTTP operations in parallel, so metadata resolution and package downloads are fast. Metadata and package files are cached on the local disk as well. `pak` has a dependency solver, so it finds version conflicts before performing the installation. This version of `pak` supports CRAN, `Bioconductor` and `GitHub` packages as well.
License GPL-3
Encoding UTF-8
LazyData true
ByteCompile true
RoxygenNote 7.2.1.9000
Depends R (>= 3.2)
Imports tools, utils
Suggests callr (>= 3.7.0), cli (>= 3.2.0), covr, curl (>= 4.3.2), desc (>= 1.4.1), digest, distro, filelock (>= 1.0.2), gitcreds, glue (>= 1.6.2), mockery, pingr, jsonlite (>= 1.8.0), pkgcache (>= 2.0.1), pkgdeends (>= 0.3.0), pkgsearch (>= 3.1.0), prettyunits, processx (>= 3.5.2), ps (>= 1.6.0), rprojroot (>= 2.0.2), rstudioapi, testthat, withr
Note This field has Remotes syntax, but repeat remotes in `Remotes`!
Config/needs/dependencies callr, desc, cli, curl, distro, filelock, glue, jsonlite, pkgcache, pkgdeends, pkgsearch, prettyunits, processx, ps, rprojroot
Config/Needs/website r-lib/asciicast,
   r-lib/roxygen2@feature/html-blocks, tidyverse/tidytemplate
Config/testthat/edition 3
URL https://pak.r-lib.org/
BugReports https://github.com/r-lib/pak/issues
BuildResaveData no
NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2022-09-08 21:30:02 UTC

R topics documented:

cache_summary .......................................................... 3
config ................................................................. 4
faq ................................................................. 6
features ............................................................. 7
handle_package_not_found ........................................... 9
install ............................................................ 10
lib_status ............................................................ 11
local_deps ........................................................... 11
local_deps_explain .................................................... 12
local_install ......................................................... 13
local_install_deps .................................................. 14
local_install_dev_deps ............................................... 15
local_package_trees .................................................. 16
local_system_requirements ......................................... 16
lockfile_create ....................................................... 18
lockfile_install .................................................... 19
meta_summary ....................................................... 19
pak ................................................................. 21
pak_cleanup ......................................................... 21
pak_install_extra .................................................... 22
pak_package_sources ................................................ 23
pak_setup ........................................................... 25
pak_sitrep .......................................................... 26
pak_update ........................................................... 26
pkg_deps ............................................................. 27
pkg_deps_explain .................................................... 28
pkg_deps_tree ......................................................... 29
pkg_download .......................................................... 29
pkg_history .......................................................... 30
pkg_install .......................................................... 31
pkg_name_check ....................................................... 32
pkg_remove ........................................................... 33
pkg_search ........................................................... 34
pkg_status ............................................................ 34
repo_add .............................................................. 35
Description
Various utilities to inspect and clean the package cache. See the pkgcache package if you need for control over the package cache.

Usage
- `cache_summary()`
- `cache_list(...)`
- `cache_delete(...)`
- `cache_clean()`

Arguments

- `...` For `cache_list()` and `cache_delete()`, `...` may contain filters, where the argument name is the column name. E.g. package, version, etc. Call `cache_list()` without arguments to see the available column names. If you call `cache_delete()` without arguments, it will delete all cached files.

Details
- `cache_summary()` returns a summary of the package cache.
- `cache_list()` lists all (by default), or a subset of packages in the package cache.
- `cache_delete()` deletes files from the cache.
- `cache_clean()` deletes all files from the cache.

Value
- `cache_summary()` returns a list with elements:
  - `cachepath`: absolute path to the package cache
  - `files`: number of files (packages) in the cache
  - `size`: total size of package cache in bytes
- `cache_list()` returns a data frame with the data about the cache.
- `cache_delete()` returns nothing.
- `cache_clean()` returns nothing.
Examples

# Summary
    cache_summary()

# List packages
    cache_list()
    cache_list(package = "recipes")
    cache_list(platform = "source")

# Delete packages
    cache_delete(package = "knitr")
    cache_delete(platform = "macos")

    cache_clean()

config  pak configuration

Description
Environment variables and options that modify the default pak behavior.

Environment variables:

PKG_BUILD_VIGNETTES: TODO
PKG_CACHE_DIR: TODO
PKG_CRAN_MIRROR: TODO
PKG_DEPENDENCIES: TODO
PKG_LIBRARY: TODO
PKG_METADATA_CACHE_DIR: TODO
PKG_METADATA_UPDATE_AFTER: TODO
PKG_PLATFORMS: TODO
PKG_R_VERSIONS:
TODO
PKG_SYSREQS:
TODO
PKG_SYSREQS_RSPM_REPO_ID:
TODO
PKG_SYSREQS_RSPM_URL:
TODO
PKG_SYSREQS_SUDO:
TODO
PKG_SYSREQS_VERBOSE:
TODO
PKG_WINDOWS_ARCHS:
TODO

Options:

Ncpus:
Set to the desired number of worker processes for package installation. If not set, then pak will use the number of logical processors in the machine.

pak.no_extra_messages:
Set to TRUE to omit pak’s warnings about missing extra packages.

pkg.build_vignettes:
TODO
pkg.cache_dir:
TODO
pkg.cran_mirror:
TODO
pkg.dependencies:
TODO
pkg.library:
TODO
pkg.metadata_cache_dir:
TODO
pkg.metadata_update_after:
TODO
pkg.package_cache_dir:
TODO
pkg.platforms:
TODO
pkg.r_versions:
TODO
pkg.sysreqs: TODO
pkg.sysreqs_rspm_repo_id: TODO
pkg.sysreqs_rspm_url: TODO
pkg.sysreqs_sudo: TODO
pkg.sysreqs_verbose: TODO
pkg.windows_archs: TODO
repos:

The CRAN-like repositories to use. See ?options for details.

---

### Description

Please look at this before asking questions.

### Package installation

**How do I reinstall a package?:**
pak does not reinstall a package, if the same version is already installed. Sometimes you still want a reinstall, e.g. to fix a broken installation. In this case you can delete the package and then install it, or use the ?reinstall parameter:

```
pak::pkg_install("tibble")
pak::pkg_install("tibble?reinstall")
```

**How do I install a package from source?:**
To force the installation of a source package (instead of a binary package), use the ?source parameter:

```
pak::pkg_install("tibble?source")
```

**How do I ignore an optional dependency?:**

```
pak::pkg_install(  
    c("tibble", "DiagrammeR?=ignore", "formattable?=ignore"),  
    dependencies = TRUE  
  )
```

The syntax is

```
<packagename>=?ignore
```

Note that you can only ignore *optional* dependencies, i.e. packages in Suggests and Enhances.
How can I use pak with renv?:
You cannot currently, but keep an eye on this issue: https://github.com/r-lib/pak/issues/343

Description
Features that make pak special.

pak is fast
Parallel HTTP:
pak performs HTTP queries concurrently. This is true when
- it downloads package metadata from package repositories,
- it resolves packages from CRAN, GitHub, URLs, etc,
- it downloads the actual package files,
- etc.

Parallel installation:
pak installs packages concurrently, as much as their dependency graph allows this.

Caching:
pak caches metadata and package files, so you don’t need to re-download the same files over and over.

pak is safe
Plan installation up front:
pak creates an installation plan before downloading any packages. If the plan is unsuccessful, then it fails without downloading any packages.

Auto-install missing dependencies:
When requesting the installation of a package, pak makes sure that all of its dependencies are also installed.

Keeping binary packages up-to-date:
pak automatically discards binary packages from the cache, if a new build of the same version is available on CRAN.

Correct CRAN metadata errors:
pak can correct some of CRAN’s metadata issues, e.g.:
- New version of the package was released since we obtained the metadata.
• macOS binary package is only available at https://mac.r-project.org/ because of a synchronization issue.

**Graceful handling of locked package DLLs on Windows:**
pak handles the situation of locked package DLLs, as well as possible. It detects which process locked them, and offers the choice of terminating these processes. It also unloads packages from the current R session as needed.

**pak keeps its own dependencies isolated:**
pak keeps its own dependencies in a private package library and never loads any packages. (Only in background processes).

**pak is convenient**

**pak comes as a self-contained binary package:**
On the most common platforms. No dependencies, no system dependencies, no compiler needed. (See also the installation manual.)

**Install packages from multiple sources:**
• CRAN, Bioconductor
• GitHub
• URLs
• Local files or directories.

**Ignore certain optional dependencies:**
pak can ignore certain optional dependencies if requested.

**CRAN package file sizes:**
pak knows the sizes of CRAN package files, so it can estimate how much data you need to download, before the installation.

**Bioconductor version detection:**
pak automatically selects the Bioconductor version that is appropriate for your R version. No need to set any repositories.

**Time travel with MRAN or RSPM:**
pak can use MRAN (Microsoft R Application Network, https://mran.microsoft.com/) or RSPM (RStudio Public Package Manager, https://packagemanager.rstudio.com/client/#/) to install from snapshots or CRAN.

**pak can install dependencies of local packages:**
Very handy for package development!
Description

Use this function to set up a global error handler, that is called if R fails to load a package. This handler will offer you the choice of installing the missing package (and all its dependencies), and in some cases it can also remedy the error and restart the code.

Usage

`handle_package_not_found(err)`

Arguments

`err` The error object, of class `packageNotFoundError`.

Details

You are not supposed to call this function directly. Instead, set it up as a global error handler, possibly in your `.Rprofile` file:

```r
if (interactive() && getRversion() >= "4.0.0") {
  globalCallingHandlers(
    packageNotFoundError = function(err) {
      try(pak::handle_package_not_found(err))
    }
  )
}
```

Global error handlers are only supported in R 4.0.0 and later.

Currently `handle_package_not_found()` does not do anything in non-interactive mode (including in knitr, testthat and RStudio notebooks), this might change in the future.

In some cases it is possible to remedy the original computation that tried to load the missing package, and pak will offer you to do so after a successful installation. Currently, in R 4.0.4, it is not possible to continue a failed `library()` call.

Value

Nothing.
install

All about installing pak.

Description

Read this if the default installation methods do not work for you or if you want the RC or development version.

Pre-built binaries:

Install a binary build of pak from our repository on GitHub:

```r
```

This is supported for the following systems:

<table>
<thead>
<tr>
<th>OS</th>
<th>CPU</th>
<th>R version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>x86_64</td>
<td>R 3.4.0 - R-devel</td>
</tr>
<tr>
<td>Linux</td>
<td>aarch64</td>
<td>R 3.4.0 - R-devel</td>
</tr>
<tr>
<td>macOS High Sierra+</td>
<td>x86_64</td>
<td>R 3.4.0 - R-devel</td>
</tr>
<tr>
<td>macOS Big Sur+</td>
<td>aarch64</td>
<td>R 4.1.0 - R-devel</td>
</tr>
<tr>
<td>Windows</td>
<td>x86_64</td>
<td>R 3.4.0 - R-devel</td>
</tr>
</tbody>
</table>

For macOS we only support the official CRAN R build. Other builds, e.g. Homebrew R, are not supported.

Install from CRAN:

Install the released version of the package from CRAN as usual:

```r
install.packages("pak")
```

This potentially needs a C compiler on platforms CRAN does not have binaries packages for.

Nightly builds:

We have nightly binary builds, for the same systems as the table above:

```r
```

stable, rc and devel streams:

We have three types of binaries available:

- **stable** corresponds to the latest CRAN release of CRAN.
- **rc** is a release candidate build, and it is available about 1-2 weeks before a release. Otherwise it is the same as the stable build.
- **devel** has builds from the development tree. Before release it might be the same as the rc build.

The streams are available under different repository URLs:

```r
stream <- "rc"
```
lib_status

Status of packages in a library

Description

Status of packages in a library

Usage

lib_status(lib = .libPaths()[1])

pkg_list(lib = .libPaths()[1])

Arguments

lib

Path to library.

Value

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

See Also

Other package functions: pak_package_sources, pak(), pkg_deps_tree(), pkg_deps(), pkg_download(), pkg_install(), pkg_remove(), pkg_status()

local_deps

Dependencies of a package tree

Description

Dependencies of a package tree

Usage

local_deps(root = ".", upgrade = TRUE, dependencies = NA)

local_deps_tree(root = ".", upgrade = TRUE, dependencies = NA)

local_dev_deps(root = ".", upgrade = TRUE, dependencies = TRUE)

local_dev_deps_tree(root = ".", upgrade = TRUE, dependencies = TRUE)
**local_deps_explain**

**Arguments**

- **root**: Path to the package tree.
- **upgrade**: Whether to use the most recent available package versions.
- **dependencies**: Which dependencies to print. Defaults to the hard dependencies for `local_deps()` and `local_deps_tree()` and the hard dependencies plus the development dependencies for `local_dev_deps()` and `local_dev_deps_tree()`.

**Value**

All of these functions return the dependencies in a data frame. `local_deps_tree()` and `local_dev_deps_tree()` also print the dependency tree.

**See Also**

Other local package trees: `local_deps_explain()`, `local_install_deps()`, `local_install_dev_deps()`, `local_install()`, `local_package_trees`, `pak()`

---

**local_deps_explain**

*Explain dependencies of a package tree*

**Description**

These functions are similar to `pkg_deps_explain()`, but work on a local package tree. `local_dev_deps_explain()` also includes development dependencies.

**Usage**

```
local_deps_explain(deps, root = ".", upgrade = TRUE, dependencies = NA)

local_dev_deps_explain(deps, root = ".", upgrade = TRUE, dependencies = TRUE)
```

**Arguments**

- **deps**: Package names of the dependencies to explain.
- **root**: Path to the package tree.
- **upgrade**: Whether to use the most recent available package versions.
- **dependencies**: Which dependencies to print. Defaults to the hard dependencies for `local_deps()` and `local_deps_tree()` and the hard dependencies plus the development dependencies for `local_dev_deps()` and `local_dev_deps_tree()`.

**See Also**

Other local package trees: `local_deps()`, `local_install_deps()`, `local_install_dev_deps()`, `local_install()`, `local_package_trees`, `pak()`
local_install

Install a package tree

Description

Installs a package tree (or source package file), together with its dependencies.

Usage

local_install(
  root = ".",
  lib = .libPaths()[1],
  upgrade = TRUE,
  ask = interactive(),
  dependencies = NA
)

Arguments

root Path to the package tree.
lib Package library to install the packages to. Note that all dependent packages will
  be installed here, even if they are already installed in another library.
upgrade When FALSE, the default, does the minimum amount of work to give you the
  latest version of pkg. It will only upgrade packages if pkg, or one of its depen-
  dencies, explicitly requires a higher version than what you currently have.
  When upgrade = TRUE, will do ensure that you have the latest version of pkg
  and all its dependencies.
ask Whether to ask for confirmation when installing a different version of a package
  that is already installed. Installations that only add new packages never require
  confirmation.
dependencies Dependency types. See pkgdepends::as_pkg_dependencies() for possible
  values. Note that changing this argument from the default might result an in-
  stallation failure, e.g. if you set it to FALSE, packages might not build if their
  dependencies are not already installed.

Details

local_install() is equivalent to pkg_install("local::.").

Value

Data frame, with information about the installed package(s).

See Also

Other local package trees: local_deps_explain(), local_deps(), local_install_deps(), local_install_dev_deps(),
local_package_trees, pak()
local_install_deps  
*Install the dependencies of a package tree*

**Description**

Installs the hard dependencies of a package tree (or source package file), without installing the package tree itself.

**Usage**

```r
local_install_deps(
  root = ".",
  lib = .libPaths()[1],
  upgrade = TRUE,
  ask = interactive(),
  dependencies = NA
)
```

**Arguments**

- **root**  
  Path to the package tree.

- **lib**  
  Package library to install the packages to. Note that all dependent packages will be installed here, even if they are already installed in another library.

- **upgrade**  
  When FALSE, the default, does the minimum amount of work to give you the latest version of pkg. It will only upgrade packages if pkg, or one of its dependencies, explicitly requires a higher version than what you currently have. When upgrade = TRUE, will do ensure that you have the latest version of pkg and all its dependencies.

- **ask**  
  Whether to ask for confirmation when installing a different version of a package that is already installed. Installations that only add new packages never require confirmation.

- **dependencies**  
  Dependency types. See `pkgdepends::as_pkg_dependencies()` for possible values. Note that changing this argument from the default might result an installation failure, e.g. if you set it to FALSE, packages might not build if their dependencies are not already installed.

**Details**

Note that development (and optional) dependencies, under Suggests in DESCRIPTION, are not installed. If you want to install them as well, use `local_install_dev_deps()`.

**Value**

Data frame, with information about the installed package(s).
local_install_dev_deps

Install all dependencies of a package tree

Description

Installs all dependencies of a package tree (or source package file), without installing the package tree itself. It installs the development dependencies as well, specified in the Suggests field of DESCRIPTION.

Usage

```r
local_install_dev_deps(
  root = ".",  
  lib = .libPaths()[1], 
  upgrade = TRUE, 
  ask = interactive(), 
  dependencies = TRUE
)
```

Arguments

- **root**: Path to the package tree.
- **lib**: Package library to install the packages to. Note that all dependent packages will be installed here, even if they are already installed in another library.
- **upgrade**: When FALSE, the default, does the minimum amount of work to give you the latest version of pkg. It will only upgrade packages if pkg, or one of its dependencies, explicitly requires a higher version than what you currently have. When upgrade = TRUE, will do ensure that you have the latest version of pkg and all its dependencies.
- **ask**: Whether to ask for confirmation when installing a different version of a package that is already installed. Installations that only add new packages never require confirmation.
- **dependencies**: Dependency types. See `pkgdepends::as_pkg_dependencies()` for possible values. Note that changing this argument from the default might result an installation failure, e.g. if you set it to FALSE, packages might not build if their dependencies are not already installed.

See Also

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_dev_deps()`, `local_install()`, `local_package_trees`, `pak()`
**Description**

pak can install packages from local package trees. This is convenient for package development. See the following functions:

- `local_install()` installs a package from a package tree and all of its (hard) dependencies (i.e. Includes, Depends, LinkingTo).
- `local_install_deps()` installs all hard dependencies of a package.
- `local_install_dev_deps()` installs all hard and soft dependencies of a package. This function is intended for active package development.

**Details**

Note that the last two functions do not install the package in the specified package tree itself, only its dependencies. This is convenient if the package itself is loaded via some other means, e.g. `devtools::load_all()`, for development.

**See Also**

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_deps()`, `local_install_dev_deps()`, `local_install()`, `pak()`

---

**local_system_requirements**

`Query system requirements`

**Description**

Returns a character vector of commands to run that will install system requirements for the queried operating system.

`local_system_requirements()` queries system requirements for a dev package (and its dependencies) given its root path.

`pkg_system_requirements()` queries system requirements for existing packages (and their dependencies).
Usage

local_system_requirements(
  os = NULL,
  os_release = NULL,
  root = ".",
  execute = FALSE,
  sudo = execute,
  echo = FALSE
)

pkg_system_requirements(
  package,
  os = NULL,
  os_release = NULL,
  execute = FALSE,
  sudo = execute,
  echo = FALSE
)

Arguments

os, os_release The operating system and operating system release version, e.g. "ubuntu", "debian", "centos", "redhat". See https://github.com/rstudio/r-system-requirements#operating-systems for all full list of supported operating systems. If NULL, the default, these will be looked up using distro::distro().

root Path to the package tree.

execute, sudo If execute is TRUE, pak will execute the system commands (if any). If sudo is TRUE, pak will prepend the commands with sudo.

echo If echo is TRUE and execute is TRUE, echo the command output.

package Package names to lookup system requirements for.

Value

A character vector of commands needed to install the system requirements for the package.

Examples

local_system_requirements("ubuntu", "20.04")

pkg_system_requirements("pak", "ubuntu", "20.04")
pkg_system_requirements("pak", "redhat", "7")
pkg_system_requirements("config", "ubuntu", "20.04") # no sys reqs
pkg_system_requirements("curl", "ubuntu", "20.04")
pkg_system_requirements("git2r", "ubuntu", "20.04")
pkg_system_requirements(c("config", "git2r", "curl"), "ubuntu", "20.04")
# queried packages must exist
lockfile_create

Create a lock file

Description

The lock file can be used later, possibly in a new R session, to carry out the installation of the dependencies, with `lockfile_install()`.

Usage

```r
lockfile_create(
  pkg = "deps::.",
  lockfile = "pkg.lock",
  lib = NULL,
  upgrade = FALSE,
  dependencies = NA
)
```

Arguments

- `pkg` Package names or remote package specifications to install. See `pak package sources` for details.
- `lockfile` Path to the lock file.
- `lib` Package library to install the packages to. Note that all dependent packages will be installed here, even if they are already installed in another library.
- `upgrade` When FALSE, the default, does the minimum amount of work to give you the latest version of pkg. It will only upgrade packages if pkg, or one of its dependencies, explicitly requires a higher version than what you currently have. When `upgrade = TRUE`, will do ensure that you have the latest version of pkg and all its dependencies.
- `dependencies` Dependency types. See `pkgdepcends::as_pkg_dependencies()` for possible values. Note that changing this argument from the default might result an installation failure, e.g. if you set it to FALSE, packages might not build if their dependencies are not already installed.

Details

Note, since the URLs of CRAN and most CRAN-like repositories change over time, in practice you cannot use the lock file much later. For example, binary packages of older package version might be deleted from the repository, breaking the URLs in the lock file.

Currently the intended use case of lock files is on CI systems, to facilitate caching. The (hash of the) lock file provides a good key for caching systems.
**lockfile_install**

*Install packages based on a lock file*

**Description**

Install a lock file that was created with `lockfile_create()`.

**Usage**

```r
lockfile_install(lockfile = "pkg.lock", lib = .libPaths()[1], update = TRUE)
```

**Arguments**

- **lockfile**: Path to the lock file.
- **lib**: Library to carry out the installation on.
- **update**: Whether to online install the packages that either not installed in `lib`, or a different version is installed for them.

**See Also**

Other lock files: `lockfile_create()`

---

**meta_summary**

*Metadata cache utilities*

**Description**

Various utilities to inspect, update and clean the metadata cache. See the pkgcache package if you need for control over the metadata cache.

**Usage**

```r
meta_summary()

meta_list(pkg = NULL)

meta_update()

meta_clean(force = FALSE)
```
Arguments

pkg    Package names, if specified then only entries for pkg are returned.
force  If FALSE, then pak will ask for confirmation.

Details

meta_summary() returns a summary of the metadata cache.
meta_list() lists all (or some) packages in the metadata database.
meta_update() updates the metadata database. You don't normally need to call this function manually, because all pak functions (e.g. pkg_install(), pkg_download(), etc.) call it automatically, to make sure that they use the latest available metadata.
meta_clean() deletes the whole metadata DB.

Value

meta_summary() returns a list with entries:

- cachepath: absolute path of the metadata cache.
- current_db: the file that contains the current metadata database. It is currently an RDS file, but this might change in the future.
- raw_files: the files that are the downloaded PACKAGES* files.
- db_files: all metadata database files.
- size: total size of the metadata cache.

meta_list() returns a data frame of all available packages in the configured repositories.
meta_update() returns nothing.
meta_clean() returns nothing

Examples

# Metadata cache summary
meta_summary()

# The current metadata DB
meta_list()
# Selected packages only
meta_list(pkg = c("shiny", "htmlwidgets"))

# Update the metadata DB
meta_update()

# Delete the metadata DB
meta_clean()
### pak

**Install the required packages**

#### Description
Install the specified packages, or the ones required by the package or project in the current working directory.

#### Usage

```r
pak(pkg = NULL, ...)
```

#### Arguments

- `pkg` Package names or remote package specifications to install. See `pak package sources` for details. If `NULL`, will install all development dependencies for the current package.
- `...` Extra arguments are passed to `pkg_install()` or `local_install_dev_deps()`.

#### Details
This is a convenience function:

- If you want to install some packages, it is easier to type than `pkg_install()`.
- If you want to install all the packages that are needed for the development of a package or project, then it is easier to type than `local_install_dev_deps()`.
- You don’t need to remember two functions to install packages, just one.

#### See Also
Other package functions: `lib_status()`, `pak_package_sources`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_deps()`, `local_install_dev_deps()`, `local_install()`, `local_package_trees`

### pak_cleanup

**Clean up pak caches**

#### Description
Clean up pak caches
pak_install_extra

Usage

pak_cleanup(
  package_cache = TRUE,
  metadata_cache = TRUE,
  pak_lib = TRUE,
  force = FALSE
)

Arguments

package_cache     Whether to clean up the cache of package files.
metadata_cache    Whether to clean up the cache of package meta data.
pak_lib           This argument is now deprecated and does nothing.
force             Do not ask for confirmation. Note that to use this function in non-interactive
                   mode, you have to specify force = FALSE.

See Also

Other pak housekeeping: pak_sitrep()

pak_install_extra                  Install all optional dependencies of pak

Description

These packages are not required for any pak functionality. They are recommended for some func-
tions that return values that are best used with these packages. E.g. many functions return data
frames, which print nicer when the pillar package is available.

Usage

pak_install_extra(upgrade = FALSE)

Arguments

upgrade     Whether to install or upgrade to the latest versions of the optional packages.

Details

Currently only one package is optional: pillar.
Description

Package sources

Standard packages

pak can install packages from various package sources. By default, a package name without the specification of its source, refers to a CRAN or Bioconductor package. pak calls these standard packages. For example:

## CRAN package
pkg_install("glue")

## BioC package
pkg_install("limma")

When considering a standard package, the calling version of R is used to determine the available source and binary packages on CRAN and the Bioconductor repositories.

The full specification of standard packages is simply

[standard::]<package>

If you know the exact source of the package, you can also write

cran::<package>
bioc::<package>

GitHub packages

pak can install packages from GitHub repositories. Any package that is specified in the user/repo notation is taken to be a GitHub package. For example:

## Package from GitHub
pkg_install("r-lib/glue")

The full specification of GitHub packages is

[<package>=][github::<username>/<repo>[/<subdir>]]
  [@<committish> | #<pull> | @[*]release]

  • <package> is the name of the package. If this is missing, the name of the package must match the name of the repository.
  • <username>: GitHub user or organization name.
pak_package_sources

- `<repo>`: repository name.
- `<subdir>`: If the R package is in a subdirectory within the repository.
- `<commitish>`: A branch name, git tag or SHA hash, to specify the branch, tag or commit to download or install.
- `<pull>`: Pull request number, to install the branch that corresponds to a pull request.
- The `@*release` string can be used to install the latest release.

Local package trees

pak can install packages from package trees. You can either use the `local_install()` function for this, or specify the `local::` package source. E.g. these are equivalent:

```r
code
local_install("/path/to/my/package")
pkg_install("local::/path/to/my/package")
```

The `local::` form is handy if you want to mix it with other package specifications, e.g. to install a local package, and another standard package:

```r
code
pkg_install(c("local://path/to/my/package", "testthat"))
```

The Remotes field

You can mark any regular dependency defined in the `Depends`, `Imports`, `Suggests` or `Enhances` fields as being installed from a remote location by adding the remote location to `Remotes` in your `DESCRIPTION` file. This will cause pak to download and install them from the specified location, instead of CRAN.

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

```r
code
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:

```r
code
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.
The package dependency solver

pak contains a package dependency solver, that makes sure that the package source and version requirements of all packages are satisfied, before starting an installation. For CRAN and BioC packages this is usually automatic, because these repositories are generally in a consistent state. If packages depend on other other package sources, however, this is not the case.

Here is an example of a conflict detected:

```r
> pak::pkg_install(c("r-lib/pkgcache@conflict", "r-lib/cli@message"))
Error: Cannot install packages:
  * Cannot install `r-lib/pkgcache@conflict`.
    - Cannot install dependency `r-lib/cli@main`
  * Cannot install `r-lib/cli@message`.
- Conflicts `r-lib/cli@message`
```

`r-lib/pkgcache@conflict` depends on the main branch of `r-lib/cli`, whereas, we explicitly requested the message branch. Since it cannot install both versions into a single library, pak quits.

When pak considers a package for installation, and the package is given with its name only, (e.g. as a dependency of another package), then the package may have any package source. This is necessary, because one R package library may contain only at most one version of a package with a given name.

pak’s behavior is best explained via an example. Assume that you are installing a local package (see below), e.g. `local::`, and the local package depends on `pkgA` and `user/pkgB`, the latter being a package from GitHub (see below), and that `pkgA` also depends on `pkgB`. Now pak must install `pkgB` and `user/pkgB`. In this case pak interprets `pkgB` as a package from any package source, instead of a standard package, so installing `user/pkgB` satisfies both requirements.

Note that that `cran::pkgB` and `user/pkgB` requirements result a conflict that pak cannot resolve. This is because the first one must be a CRAN package, and the second one must be a GitHub package, and two different packages with the same cannot be installed into an R package library.

See Also

Other package functions: `lib_status()`, `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

---

**pak_setup**

Set up private pak library (deprecated)

---

**Description**

This function is deprecated and does nothing. Recent versions of pak do not need a `pak_setup()` call.

**Usage**

`pak_setup(mode = c("auto", "download", "copy"), quiet = FALSE)"
Arguments

mode  Where to get the packages from. "download" will try to download them from CRAN. "copy" will try to copy them from your current "regular" package library. "auto" will try to copy first, and if that fails, then it tries to download.

quiet  Whether to omit messages.

Value

The path to the private library, invisibly.

pak_sitrep  pak SITuation REPort

Description

It prints

• pak version,
• the current library path,
• location of the private library,
• whether the pak private library exists,
• whether the pak private library is functional.

Usage

pak_sitrep()

See Also

Other pak housekeeping: pak_cleanup()

pak_update  Update pak itself

Description

Use this function to update the released or development version of pak.

Usage

pak_update(force = FALSE, stream = c("auto", "stable", "rc", "devel"))
pkg_deps

Arguments

force Whether to force an update, even if no newer version is available.
stream Whether to update to the
• "stable",
• "rc" (release candidate) or
• "devel" (development) version.
• "auto" updates to the same stream as the current one.
Often there is no release candidate version, then "rc" also installs the stable version.

Value

Nothing.

pkg_deps(pkg, upgrade = TRUE, dependencies = NA)

Usage

pkg_deps(pkg, upgrade = TRUE, dependencies = NA)

Arguments

pkg Package name or remote package specification to resolve.
upgrade Whether to use the most recent available package versions.
dependencies Dependency types. See pkgdepends::as_pkg_dependencies() for possible values.

Value

A data frame.

See Also

Other package functions: lib_status(), pak_package_sources, pak(), pkg_deps_tree(), pkg_download(), pkg_install(), pkg_remove(), pkg_status()

Examples

pkg_deps("curl")
pkg_deps("r-lib/fs")
pkg_deps_explain

**pkg_deps_explain** *Explain how a package depends on other packages*

Description

Extract dependency chains from pkg to deps.

Usage

```r
pkg_deps_explain(pkg, deps, upgrade = TRUE, dependencies = NA)
```

Arguments

- `pkg` Package name or remote package specification.
- `deps` Package names of the dependencies to explain.
- `upgrade` Whether to use the most recent available package versions.
- `dependencies` Dependency types. See `pkgdepends::as_pkg_dependencies()` for possible values.

Details

This function is similar to `pkg_deps_tree()`, but its output is easier to read if you are only interested in certain packages (deps).

Value

A named list with a print method. First entries are the function arguments: pkg, deps, dependencies, the last one is `paths` and it contains the results in a named list, the names are the package names in deps.

Examples

```r
## Not run:
# How does the GH version of usethis depend on cli and ps?
pkg_deps_explain("r-lib/usethis", c("cli", "ps"))

## End(Not run)
```
**pkg_deps_tree**

Draw the dependency tree of a package

**Description**

Draw the dependency tree of a package

**Usage**

```r
pkg_deps_tree(pkg, upgrade = TRUE, dependencies = NA)
```

**Arguments**

- `pkg`: Package name or remote package specification to resolve.
- `upgrade`: Whether to use the most recent available package versions.
- `dependencies`: Dependency types. See `pkgdepends::as_pkg_dependencies()` for possible values.

**Value**

The same data frame as `pkg_deps()`, invisibly.

**See Also**

Other package functions: `lib_status()`, `pak_package_sources`, `pak()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

**Examples**

```r
pkg_deps_tree("dplyr")
pkg_deps_tree("r-lib/usethis")
```

**pkg_download**

Download a package and potentially its dependencies as well

**Description**

Download a package and potentially its dependencies as well
pkg_download(  
  pkg,  
  dest_dir = ".",  
  dependencies = FALSE,  
  platforms = NULL,  
  r_versions = NULL  
)

pkg_history(pkg)

**Usage**

pkg_download(  
  pkg,  
  dest_dir = ".",  
  dependencies = FALSE,  
  platforms = NULL,  
  r_versions = NULL  
)

**Arguments**

- **pkg**: Package names or remote package specifications to download.
- **dest_dir**: Destination directory for the packages. If it does not exist, then it will be created.
- **dependencies**: Dependency types, to download the (recursive) dependencies of `pkg` as well. See `pkgdepends::as_pkg_dependencies()` for possible values.
- **platforms**: Types of binary or source packages to download. The default is the value of `pkgdepends::default Platforms()`.
- **r_versions**: R version(s) to download packages for. (This does not matter for source packages, but it does for binaries.) It defaults to the current R version.

**Value**

Data frame with information about the downloaded packages, invisibly.

**See Also**

Other package functions: `lib_status()`, `pak_package_sources`, `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

**Examples**

```r
pkg_download("forcats")
pkg_download("r-lib/pak", platforms = "source")
```

**pkg_history**

*Query the history of a package*

**Description**

Query the history of a package

**Usage**

`pkg_history(pkg)`
**pkg_install**

Arguments

pkg  
Package name.

Value

A data frame, with one row per package version.

```
pkg_install(pkg)  
```

**Install a package**

Description

Install a package and its dependencies into a single package library.

Usage

```
pkg_install(
  pkg,
  lib = .libPaths()[[1L]],
  upgrade = FALSE,
  ask = interactive(),
  dependencies = NA
)
```

Arguments

pkg  
Package names or remote package specifications to install. See pak package sources for details.

lib  
Package library to install the packages to. Note that all dependent packages will be installed here, even if they are already installed in another library.

upgrade  
When FALSE, the default, does the minimum amount of work to give you the latest version of pkg. It will only upgrade packages if pkg, or one of its dependencies, explicitly requires a higher version than what you currently have.

When upgrade = TRUE, will do ensure that you have the latest version of pkg and all its dependencies.

ask  
Whether to ask for confirmation when installing a different version of a package that is already installed. Installations that only add new packages never require confirmation.

dependencies  
Dependency types. See pkgdepends::as_pkg_dependencies() for possible values. Note that changing this argument from the default might result an installation failure, e.g. if you set it to FALSE, packages might not build if their dependencies are not already installed.

Value

(Invisibly) A data frame with information about the installed package(s).
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.

Examples
```r
## Not run:
pkg_install("dplyr")

# Upgrade dplyr and all its dependencies
pkg_install("dplyr", upgrade = TRUE)

# Install the development version of dplyr
pkg_install("tidyverse/dplyr")

# Switch back to the CRAN version. This will be fast because
# pak will have cached the prior install.
pkg_install("dplyr")

## End(Not run)
```
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.

pkg_remove Remove installed packages

Description
Remove installed packages

Usage
pkg_remove(pkg, lib = .libPaths()[[1L]])

Arguments
pkg A character vector of packages to remove.
lib library to remove packages from

See Also
Other package functions: lib_status(), pak_package_sources, pak(), pkg_deps_tree(), pkg_deps(), pkg_download(), pkg_install(), pkg_status()
pkg_search

Search CRAN packages

Description

Search the indexed database of current CRAN packages. It uses the pkgsearch package. See that package for more details and also pkgsearch::pkg_search() for pagination, more advanced searching, etc.

Usage

pkg_search(query, ...)

Arguments

query Search query string.
...

Additional arguments passed to pkgsearch::pkg_search()

Value

A data frame, that is also a pak_search_result object with a custom print method. To see the underlying table, you can use [] to drop the extra classes. See examples below.

Examples

# Simple search
pkg_search("survival")

# See the underlying data frame
psro <- pkg_search("ropensci")
psro[

pkg_status

Display installed locations of a package

Description

Display installed locations of a package

Usage

pkg_status(pkg, lib = .libPaths())
repo_add

Arguments

pkg       Name of one or more installed packages to display status for.
lib       One or more library paths to lookup packages status in.

Value

Data frame with data about installations of pkg. Columns include: library, package, title, version.

See Also

Other package functions: `lib_status()`, `pak_package_sources`, `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`

Examples

```r
## Not run:
pkg_status("MASS")

## End(Not run)
```

text

repo_add

Add a new CRAN-like repository

Description

Add a new repository to the list of repositories that pak uses to look for packages.

Usage

```r
repo_add(..., .list = NULL)

repo_resolve(spec)
```

Arguments

...       Repository specifications, possibly named character vectors. See details below.
.list     List or character vector of repository specifications. This argument is easier to use programmatically than ... See details below.
.spec     Repository specification, a possibly named character scalar.

Details

`repo_add()` adds new repositories. It resolves the specified repositories using `repo_resolve()` and then modifies the repos global option.

`repo_add()` only has an effect in the current R session. If you want to keep your configuration between R sessions, then set the repos option to the desired value in your user or project .Rprofile file.
Value

repo_resolve() returns a named character scalar, the URL of the repository.

Repository specifications

The format of a repository specification is a named or unnamed character scalar. If the name is missing, pak 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
- RSPM@<date>, RSPM (RStudio Package Manager) snapshot, at the specified date.
- RSPM@<package>-<version> RSPM snapshot, for the day after the release of <version> of <package>.
- RSPM@R-<version> RSPM snapshot, for the day after R <version> was released.
- 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 RSPM at https://packagemanager.rstudio.com/client/#/.
- See more about MRAN snapshots at https://mran.microsoft.com/timemachine.
- All dates (or times) can be specified in the ISO 8601 format.
- If RSPM 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 RSPM snapshot will trigger an error.
- Dates before the first, or after the last MRAN snapshot will trigger an error.
- Unknown R or package versions will trigger an error.

See Also

Other repository functions: repo_get(), repo_status()

Examples

repo_add(RSPM@dplyr100 = "RSPM@dplyr-1.0.0")
repo_get()

repo_resolve("MRAN@2020-01-21")
repo_resolve("RSPM@2020-01-21")
repo_resolve("MRAN@dplyr-1.0.0")
repo_resolve("RSPM@dplyr-1.0.0")
repo_resolve("MRAN@R-4.0.0")
repo_resolve("RSPM@R-4.0.0")
repo_get

Query the currently configured CRAN-like repositories

Description
pak uses the repos option, see options(). It also automatically adds a CRAN mirror if none is set up, and the correct version of the Bioconductor repositories. See the cran_mirror and bioc arguments.

Usage
repo_get(r_version = getRversion(), bioc = TRUE, cran_mirror = NULL)

Arguments
- r_version: R version to use to determine the correct Bioconductor version, if bioc = TRUE.
- bioc: Whether to automatically add the Bioconductor repositories to the result.
- cran_mirror: CRAN mirror to use. Leave it at NULL to use the mirror in getOption("repos") or an automatically selected one.

Details
repo_get() returns the table of the currently configured repositories.

See Also
Other repository functions: repo_add(), repo_status()

Examples
repo_get()

repo_status
Show the status of CRAN-like repositories

Description
It checks the status of the configured or supplied repositories.
Usage

```r
repo_status(
  platforms = NULL,
  r_version = getRversion(),
  bioc = TRUE,
  cran_mirror = NULL
)

repo_ping(
  platforms = NULL,
  r_version = getRversion(),
  bioc = TRUE,
  cran_mirror = NULL
)
```

Arguments

- **platforms**: Platforms to use, default is the current platform, plus source packages.
- **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`.
- **cran_mirror**: The CRAN mirror to use.

Details

`repo_ping()` is similar to `repo_status()` but also prints a short summary of the data, and it returns its result invisibly.

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.
- **url**: base URL of the repository.
- **bioc_version**: Bioconductor version, or NA for non-Bioconductor repositories.
- **platform**: platform, possible values are source, macos and windows currently.
- **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_add()`, `repo_get()`

Examples

```r
repo_status()
repo_status(
  platforms = c("windows", "macos"),
  r_version = c("4.0", "4.1")
)
repo_ping()
```

---

**tl dr**

Simplified manual. Start here!

Description

You don’t need to read long manual pages for a simple task. This manual page collects the most common `pak` use cases.

Package installation

**Install a package from CRAN or Bioconductor:**

```
pak::pkg_install("tibble")
```

**Install a package from GitHub:**

```
pak::pkg_install("tidyverse/tibble")
```

**Install a package from a URL:**

```
pak::pkg_install("url::https://cran.r-project.org/src/contrib/Archive/tibble/tibble_3.1.7.tar.gz")
```

Package updates

**Update a package:**

```
pak::pkg_install("tibble")
```

**Update all dependencies of a package:**

```
pak::pkg_install("tibble", upgrade = TRUE)
```
Dependency lookup

- Dependencies of a CRAN or Bioconductor package:
  pak::pkg_deps("tibble")

- Dependency tree of a CRAN / Bioconductor package:
  pak::pkg_deps_tree("tibble")

- Dependency tree of a package on GitHub:
  pak::pkg_deps_tree("tidyverse/tibble")

- Dependency tree of the package in the current directory:
  pak::local_deps_tree("tibble")
  (Assuming package is in directory tibble.)

- Explain a recursive dependency:
  pak::pkg_deps_explain("tibble", "rlang")

Package development

- Install dependencies of local package:
  pak::local_install_deps()

- Install local package:
  pak::local_install()

- Install all dependencies of local package:
  pak::local_install_dev_deps()

Repositories

- List current repositories:
  pak::repo_get()

- Add custom repository:
  pak::repo_add(rhub = 'https://r-hub.r-universe.dev')
  pak::repo_get()

- Remove custom repositories:
  options(repos = getOption("repos")["CRAN"])
  pak::repo_get()

  Keeps only CRAN and (by default) Bioconductor.
Time travel using RSPM:

pak::repo_add(CRAN = "RSPM@2022-06-30")
pak::repo_get()

Time travel using MRAN:

pak::repo_add(CRAN = "MRAN@2022-06-30")
pak::repo_get()

Caches

Inspect metadata cache:

pak::meta_list()

Update metadata cache:

pak::meta_update()

Clean metadata cache:

pak::meta_clean(force = TRUE)
pak::meta_summary()

Inspect package cache:

pak::cache_list()

View a package cache summary:

pak::cache_summary()

Clean package cache:

pak::cache_clean()

Libraries

List packages in a library:

pak::lib_status(Sys.getenv("R_LIBS_USER"))

Pass the directory of the library as the argument.
pak_cleanup, 21, 26
pak_install_extra, 22
pak_package_sources, 11, 21, 23, 27, 29, 30, 32, 33, 35
pak_setup, 25
pak_sitrep, 22, 26
pak_sitrep_data (pak_sitrep), 26
pak_update, 26
pkg_deps, 11, 21, 25, 27, 29, 30, 32, 33, 35
pkg_deps(), 29
pkg_deps_explain, 28
pkg_deps_explain(), 12
pkg_deps_tree, 11, 21, 25, 27, 29, 30, 32, 33, 35
pkg_deps_tree(), 28
pkg_download, 11, 21, 25, 27, 29, 30, 32, 33, 35
pkg_download(), 20
pkg_history, 30
pkg_install, 11, 21, 25, 27, 29, 30, 31, 33, 35
pkg_install(), 20, 21
pkg_list (lib_status), 11
pkg_name_check, 32
pkg_remove, 11, 21, 25, 27, 29, 30, 32, 33, 35
pkg_search, 34
pkg_status, 11, 21, 25, 27, 29, 30, 32, 33, 34
pkg_system_requirements
  (local_system_requirements), 16
pkgdepends::as_pkg_dependencies(), 13–15, 18, 27–31
pkgdepends::default_platforms(), 30
pkgsearch::pkg_search(), 34
repo_add, 35, 37, 39
repo_get, 36, 37, 39
repo_ping (repo_status), 37
repo_resolve (repo_add), 37
repo_status, 36, 37, 37
tldr, 39