Package 'Require'

September 10, 2020

Type Package

Title Installing and Loading R Packages for Reproducible Workflows

Description A single key function, 'Require' that wraps 'install.packages', 'remotes::install_github', 'versions::install.versions', and 'base::require' that allows for reproducible workflows. As with other functions in a reproducible workflow, this package emphasizes functions that return the same result whether it is the first or subsequent times running the function. Maturing.

URL https://Require.predictiveecology.org, https://github.com/PredictiveEcology/Require

Date 2020-09-08

Version 0.0.8

Depends R (>= 3.5)

Imports data.table (>= 1.10.4), methods, remotes, utils

Suggests testit

Encoding UTF-8

Language en-CA

License GPL-3

BugReports https://github.com/PredictiveEcology/Require/issues

ByteCompile yes

RoxygenNote 7.1.1

NeedsCompilation no

Author Eliot J B McIntire [aut, cre] (<https://orcid.org/0000-0002-6914-8316>), Her Majesty the Queen in Right of Canada, as represented by the Minister of Natural Resources Canada [cph]

Maintainer Eliot J B McIntire <eliot.mcintire@canada.ca>

Repository CRAN

Date/Publication 2020-09-10 06:30:16 UTC

R topics documented:

`Require-package` *Require: Installing and Loading R Packages for Reproducible Workflows*

Description

A single key function, 'Require' that wraps 'install.packages', 'remotes::install_github', 'versions::install.versions', and 'base::require' that allows for reproducible workflows. As with other functions in a reproducible workflow, this package emphasizes functions that return the same result whether it is the first or subsequent times running the function.

This is an "all in one" function that will run `install.packages` for CRAN packages, `remotes::install_github` for `https://github.com/` packages and will install specific versions of each package if versions are specified either via an (in)equality (e.g., `"Holidays (>=1.0.0)"` or `"Holidays (==1.0.0)"` for an exact version) or with a `packageVersionFile`. If `require = TRUE`, the default, the function will then run `require` on all named packages that satisfy their version requirements. If packages are already installed (`packages` supplied), and their optional version numbers are satisfied, then the "install" component will be skipped.

Usage

```
Require(
  packages,
  packageVersionFile,
  libPaths,
  install_githubArgs = list(),
  install.packagesArgs = list(),
```

```
  standAlone = getOption("Require.standAlone", FALSE),
  install = getOption("Require.install", TRUE),
  require = getOption("Require.require", TRUE),
  repos = getOption("repos"),
  purge = getOption("Require.purge", FALSE),
  verbose = getOption("Require.verbose", FALSE),
  ...
)
```

Arguments

packages	Character vector of packages to install via `install.packages`, then load (i.e., with `library`). If it is one package, it can be unquoted (as in `require`). In the case of a GitHub package, it will be assumed that the name of the repository is the name of the package. If this is not the case, then pass a named character vector here, where the names are the package names that could be different than the GitHub repository name.
packageVersionFile	If provided, then this will override all `install.package` calls with `versions::install.versions`
libPaths	The library path (or libraries) where all packages should be installed, and looked for to load (i.e., call `library`). This can be used to create isolated, stand alone package installations, if used with `standAlone = TRUE`. Currently, the path supplied here will be prepended to `.libPaths()` (temporarily during this call) to `Require` if `standAlone = FALSE` or will set (temporarily) `.libPaths()` to `c(libPaths,tail(libPaths(),1)` to keep base packages.
install_githubArgs	List of optional named arguments, passed to `install_github`.
install.packagesArgs	List of optional named arguments, passed to `install.packages`.
standAlone	Logical. If `TRUE`, all packages will be installed to and loaded from the `libPaths` only. If `FALSE`, then `libPath` will be prepended to `.libPaths()` during the `Require` call, resulting in shared packages, i.e., it will include the user's default package folder(s). This can be create dramatically faster installs if the user has a substantial number of the packages already in their personal library. Default `FALSE` to minimize package installing.
install	Logical or "force". If `FALSE`, this will not try to install anything. If `"force"`, then it will force installation of requested packages, mimicking a call to e.g., `install.packages`. If `TRUE`, the default, then this function will try to install any missing packages or dependencies.
require	Logical. If `TRUE`, the default, then the function will attempt to call `require` on all requested packages, possibly after they are installed.
repos	The remote repository (e.g., a CRAN mirror), passed to either `install.packages`, `install_github` or `installVersions`.
purge	Logical. Should all caches be purged Default is `getOption("Require.purge",FALSE)`. There is a lot of internal caching of results throughout the `Require` package. These help with speed and reduce calls to internet sources. However, sometimes

these caches must be purged. The cached values are renewed when found to be too old, with the age limit. This maximum age can be set in seconds with the environment variable R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE, or if unset, defaults to 3600 (one hour – see available.packages).

Internally, there are calls to available.packages

verbose — Numeric. If 1 (less) or 2 (more), there will be a data.table with many details attached to the output

... — Passed to *all* of install_github, install.packages, and remotes::install_version, i.e., the function will error if all of these functions can not use the ... argument. Good candidates are e.g., type or dependencies. This can be used with install_githubArgs or install.packageArgs which give individual options for those 2 internal function calls.

Details

standAlone will either put the Required packages and their dependencies *all* within the libPaths (if TRUE) or if FALSE will only install packages and their dependencies that are otherwise not installed in .libPaths()[1], i.e., the current active R package directory. Any packages or dependencies that are not yet installed will be installed in libPaths.

GitHub Package

Follows remotes::install_github standard as this is what is used internally. As with remotes::install_github, it is not possible to specify a past version of a GitHub package, without supplying a SHA that had that package version. Similarly, if a developer does a local install e.g., via devtools::install, of an active project, this package will not be able know of the GitHub state, and thus pkgSnapshot will not be able to recover this state as there is no SHA associated with a local installation. Use Require or install_github to create a record of the GitHub state.

Package Snapshots

To build a snapshot of the desired packages and their versions, first run Require with all packages, then pkgSnapshot. If a libPaths is used, it must be used in both functions.

Mutual Dependencies

This function works best if all required packages are called within one Require call, as all dependencies can be identified together, and all package versions will be addressed (if there are no conflicts), allowing a call to pkgSnapshot to take a snapshot or "record" of the current collection of packages and versions.

Local Cache of Packages

When installing new packages, 'Require' will put all source and binary files in 'getOption("Require.RPackageCache")' whose default is 'NULL', meaning *do not cache packages locally*, and will reuse them if needed. To turn on this feature, set 'option("Require.RPackageCache" = "someExistingFolder")'.

Note

For advanced use and diagnosis, the user can set `verbose = TRUE` or 1 or 2 (or via `options("Require.verbose")`). This will attache an attribute `attr(obj,"Require")` to the output of this function.

Author(s)

Maintainer: Eliot J B McIntire <eliot.mcintire@canada.ca> (ORCID)

Other contributors:

- Her Majesty the Queen in Right of Canada, as represented by the Minister of Natural Resources Canada [copyright holder]

See Also

Useful links:

- https://Require.predictiveecology.org
- https://github.com/PredictiveEcology/Require
- Report bugs at https://github.com/PredictiveEcology/Require/issues

Examples

```
## Not run:
# simple usage, like conditional install.packages then library
library(Require)
Require("stats") # analogous to require(stats), but it checks for
                 #   pkg dependencies, and installs them, if missing
tempPkgFolder <- file.path(tempdir(), "Packages")

# use standAlone, means it will put it in libPaths, even if it already exists
#   in another local library (e.g., personal library)
Require("crayon", libPaths = tempPkgFolder, standAlone = TRUE)

# make a package version snapshot of installed packages
packageVersionFile <- "_.packageVersionTest.txt"
(pkgSnapshot(libPath = tempPkgFolder, packageVersionFile, standAlone = TRUE))

# Restart R -- to remove the old temp folder (it disappears with restarting R)
library(Require)
tempPkgFolder <- file.path(tempdir(), "Packages")
packageVersionFile <- "_.packageVersionTest.txt"
# Reinstall and reload the exact version from previous
Require(packageVersionFile = packageVersionFile, libPaths = tempPkgFolder, standAlone = TRUE)

# Create mismatching versions -- desired version is older than current installed
# This will try to install the older version, overwriting the newer version
desiredVersion <- data.frame(instPkgs="crayon", instVers = "1.3.2", stringsAsFactors = FALSE)
write.table(file = packageVersionFile, desiredVersion, row.names = FALSE)
newTempPkgFolder <- file.path(tempdir(), "Packages2")

# Note this will install the 1.3.2 version (older that current on CRAN), but
```

```
#  because crayon is still loaded in memory, it will return TRUE, using the current version
#   of crayon. To start using the older 1.3.2, need to unload or restart R
Require("crayon", packageVersionFile = packageVersionFile,
        libPaths = newTempPkgFolder, standAlone = TRUE)

# restart R again to get access to older version
# run again, this time, correct "older" version installs in place of newer one
library(Require)
packageVersionFile <- "_.packageVersionTest.txt"
newTempPkgFolder <- file.path(tempdir(), "Packages3")
Require("crayon", packageVersionFile = packageVersionFile,
        libPaths = newTempPkgFolder, standAlone = TRUE)

# Mutual dependencies, only installs once -- e.g., httr
tempPkgFolder <- file.path(tempdir(), "Packages")
Require(c("cranlogs", "covr"), libPaths = tempPkgFolder, standAlone = TRUE)

##############################################################################################
# Isolated projects -- Just use a project folder and pass to libPaths or set .libPaths() #
##############################################################################################
# GitHub packages -- restart R because crayon is needed
library(Require)
ProjectPackageFolder <- file.path(tempdir(), "ProjectA")
#  THIS ONE IS LARGE -- > 100 dependencies -- use standAlone = FALSE to
#    reuse already installed packages --> this won't allow as much control
#    of package versioning
Require("PredictiveEcology/SpaDES@development",
        libPaths = ProjectPackageFolder, standAlone = FALSE)

# To keep totally isolated: use standAlone = TRUE
#   --> setting .libPaths() directly means standAlone is not necessary; it will only
#   use .libPaths()
library(Require)
ProjectPackageFolder <- file.path("~", "ProjectA")
setLibPaths(ProjectPackageFolder)
Require("PredictiveEcology/SpaDES@development") # the latest version on GitHub
Require("PredictiveEcology/SpaDES@23002b2a92a92df4ccba7f51cdd82798800b2fa7")
                             # a specific commit (by using the SHA)


##############################################################################
# Mixing and matching GitHub, CRAN, with and without version numbering
##############################################################################
# Restart R -- when installing/loading packages, start fresh
pkgs <- c("Holidays (<=1.0.4)", "TimeWarp (<= 1.0.3)", "glmm (<=1.3.0)",
          "achubaty/amc@development", "PredictiveEcology/LandR@development (>=0.0.1)",
         "PredictiveEcology/LandR@development (>=0.0.2)", "ianmseddy/LandR.CS (<=0.0.1)")
Require::Require(pkgs)

##############################################################################
# Using libPaths -- This will only be used inside this function;
# To change .libPaths() for the whole session use a manually call to
# setLibPaths(newPath) first
```

```
################################################################################
Require::Require("SpaDES", libPaths = "~/TempLib2", standAlone = FALSE)

################################################################################
# Persistent separate packages
################################################################################
setLibPaths("~/TempLib2", standAlone = TRUE)
Require::Require("SpaDES") # not necessary to specifify standAlone here because .libPaths are set

## End(Not run)
```

---

checkPath *Check directory path*

Description

Checks the specified path to a directory for formatting consistencies, such as trailing slashes, etc.

Usage

```
checkPath(path, create)

## S4 method for signature 'character,logical'
checkPath(path, create)

## S4 method for signature 'character,missing'
checkPath(path)

## S4 method for signature '`NULL`,ANY'
checkPath(path)

## S4 method for signature 'missing,ANY'
checkPath()
```

Arguments

path	A character string corresponding to a directory path.
create	A logical indicating whether the path should be created if it does not exist. Default is FALSE.

Value

Character string denoting the cleaned up filepath.

Note

This will not work for paths to files. To check for existence of files, use `file.exists`. To normalize a path to a file, use `normPath` or `normalizePath`.

See Also

file.exists, dir.create.

Examples

```
## normalize file paths
paths <- list("./aaa/zzz",
              "./aaa/zzz/",
              ".//aaa//zzz",
              ".//aaa//zzz/",
              ".\\\\aaa\\\\zzz",
              ".\\\\aaa\\\\zzz\\\\",
              file.path(".", "aaa", "zzz"))

checked <- normPath(paths)
length(unique(checked)) ## 1; all of the above are equivalent

## check to see if a path exists
tmpdir <- file.path(tempdir(), "example_checkPath")

dir.exists(tmpdir) ## FALSE
tryCatch(checkPath(tmpdir, create = FALSE), error = function(e) FALSE) ## FALSE

checkPath(tmpdir, create = TRUE)
dir.exists(tmpdir) ## TRUE

unlink(tmpdir, recursive = TRUE)
```

DESCRIPTIONFileVersionV

GitHub package tools

Description

A series of helpers to access and deal with GitHub packages

Usage

```
DESCRIPTIONFileVersionV(file, purge = getOption("Require.purge", FALSE))

DESCRIPTIONFileOtherV(file, other = "RemoteSha")

getGitHubDESCRIPTION(pkg, purge = getOption("Require.purge", FALSE))
```

Arguments

file	A file path to a DESCRIPTION file

purge	Logical. Should all caches be purged Default is `getOption("Require.purge",FALSE)`. There is a lot of internal caching of results throughout the `Require` package. These help with speed and reduce calls to internet sources. However, sometimes these caches must be purged. The cached values are renewed when found to be too old, with the age limit. This maximum age can be set in seconds with the environment variable `R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE`, or if unset, defaults to 3600 (one hour – see `available.packages`). Internally, there are calls to `available.packages`
other	Any other keyword in a DESCRIPTION file that preceeds a ":". The rest of the line will be retrieved.
pkg	A character string with a GitHub package specification (c.f. remotes)

Details

`getGitHubDESCRIPTION` retrieves the DESCRIPTION file from GitHub.com

`detachAll` *Detach and unload all packages*

Description

This uses `pkgDepTopoSort` internally so that the package dependency tree is determined, and then packages are unloaded in the reverse order. Some packages don't unload successfully for a variety of reasons. Several known packages that have this problem are identified internally and *not* unloaded. Currently, these are `glue`, `rlang`, `ps`, `ellipsis`, and, `processx`.

Usage

```
detachAll(pkgs, dontTry = NULL, doSort = TRUE)
```

Arguments

pkgs	A character vector of packages to detach. Will be topologically sorted unless `doSort` is `FALSE`.
dontTry	A character vector of packages to not try. This can be used by a user if they find a package fails in attempts to unload it, e.g., "ps"
doSort	If `TRUE` (the default), then the `pkgs` will be topologically sorted. If `FALSE`, then it won't. Useful if the `pkgs` are already sorted.

Value

A numeric named vector, with names of the packages that were attempted. `2` means the package was successfully unloaded, `1` it was tried, but failed, `3` it was in the search path and was detached and unloaded.

extractPkgName *Extract info from package character strings*

Description

Cleans a character vector of non-package name related information (e.g., version)

Usage

```
extractPkgName(pkgs)

extractVersionNumber(pkgs)

extractInequality(pkgs)

extractPkgGitHub(pkgs)
```

Arguments

pkgs	A character string vector of packages with or without GitHub path or versions

Value

Just the package names without extraneous info.

See Also

trimVersionNumber

Examples

```
extractPkgName("Require (>=0.0.1)")
extractVersionNumber(c("Require (<=0.0.1)", "PredictiveEcology/Require@development (<=0.0.4)"))
extractInequality("Require (<=0.0.1)")
extractPkgGitHub("PredictiveEcology/Require")
```

getPkgVersions *Internals used by* Require

Description

While these are not intended to be called manually by users, they may be of some use for advanced users.

Usage

```
getPkgVersions(pkgDT, install = TRUE)

getAvailable(pkgDT, purge = FALSE, repos = getOption("repos"))

installFrom(pkgDT, purge = FALSE, repos = getOption("repos"))

doInstalls(
  pkgDT,
  install_githubArgs,
  install.packagesArgs,
  install = TRUE,
  repos = getOption("repos"),
  ...
)

doLoading(pkgDT, require = TRUE, ...)

archiveVersionsAvailable(package, repos)
```

Arguments

pkgDT	A character string with full package names or a data.table with at least 2 columns "Package" and "packageFullName".
install	Logical or "force". If FALSE, this will not try to install anything. If "force", then it will force installation of requested packages, mimicking a call to e.g., install.packages. If TRUE, the default, then this function will try to install any missing packages or dependencies.
purge	Logical. Should all caches be purged Default is getOption("Require.purge",FALSE). There is a lot of internal caching of results throughout the Require package. These help with speed and reduce calls to internet sources. However, sometimes these caches must be purged. The cached values are renewed when found to be too old, with the age limit. This maximum age can be set in seconds with the environment variable R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE, or if unset, defaults to 3600 (one hour – see available.packages). Internally, there are calls to available.packages
repos	The remote repository (e.g., a CRAN mirror), passed to either install.packages, install_github or installVersions.
install_githubArgs	List of optional named arguments, passed to install_github.
install.packagesArgs	List of optional named arguments, passed to install.packages.
...	Passed to *all* of install_github, install.packages, and remotes::install_version, i.e., the function will error if all of these functions can not use the ... argument. Good candidates are e.g., type or dependencies. This can be used with install_githubArgs or install.packageArgs which give individual options for those 2 internal function calls.

require	Logical. If TRUE, the default, then the function will attempt to call require on all requested packages, possibly after they are installed.
package	A single package name (without version or github specifications)

Details

doInstall is a wrapper around install.packages, remotes::install_github, and remotes::install_version.

doLoading is a wrapper around require.

archiveVersionsAvailable searches CRAN Archives for available versions. It has been borrowed from a sub-set of the code in a non-exported function: remotes:::download_version_url

Value

In general, these functions return a data.table with various package information, installation status, version, available version etc.

---

invertList *Invert a 2-level list*

Description

This is a simple version of purrr::transpose, only for lists with 2 levels.

Usage

```
invertList(l)
```

Arguments

l	A list with 2 levels. If some levels are absent, they will be NULL

Value

A list with 2 levels deep, inverted from l

Examples

```
# create a 2-deep, 2 levels in first, 3 levels in second
a <- list(a = list(d = 1, e = 2:3, f = 4:6), b = list(d = 5, e = 55))
invertList(a) # creates 2-deep, now 3 levels outer --> 2 levels inner
```

messageDF *Use message to print a clean square data structure*

Description

Sends to message, but in a structured way so that a data.frame-like can be cleanly sent to messaging.

Usage

```
messageDF(df, round)
```

Arguments

df	A data.frame, data.table, matrix
round	An optional numeric to pass to round

normPath *Normalize filepath*

Description

Checks the specified filepath for formatting consistencies: 1) use slash instead of backslash; 2) do tilde etc. expansion; 3) remove trailing slash.

Usage

```
normPath(path)

## S4 method for signature 'character'
normPath(path)

## S4 method for signature 'list'
normPath(path)

## S4 method for signature '`NULL`'
normPath(path)

## S4 method for signature 'missing'
normPath()
```

Arguments

path	A character vector of filepaths.

Value

Character vector of cleaned up filepaths.

Examples

```
## normalize file paths
paths <- list("./aaa/zzz",
              "./aaa/zzz/",
              ".//aaa//zzz",
              ".//aaa//zzz/",
              ".\\\\aaa\\\\zzz",
              ".\\\\aaa\\\\zzz\\\\",
              file.path(".", "aaa", "zzz"))

checked <- normPath(paths)
length(unique(checked)) ## 1; all of the above are equivalent

## check to see if a path exists
tmpdir <- file.path(tempdir(), "example_checkPath")

dir.exists(tmpdir) ## FALSE
tryCatch(checkPath(tmpdir, create = FALSE), error = function(e) FALSE) ## FALSE

checkPath(tmpdir, create = TRUE)
dir.exists(tmpdir) ## TRUE

unlink(tmpdir, recursive = TRUE)
```

---

parseGitHub *GitHub specific helpers*

Description

`install_githubV` is a vectorized `remotes::install_github`. This will attempt to identify all dependencies of all supplied packages first, then load the packages in the correct order so that each of their dependencies are met before each is installed.

Usage

```
parseGitHub(pkgDT)

install_githubV(gitPkgNames, install_githubArgs = list(), dots = dots)
```

Arguments

`pkgDT`	A character string with full package names or a data.table with at least 2 columns `"Package"` and `"packageFullName"`.
`gitPkgNames`	Character vector of package to install from GitHub

install_githubArgs
: Any arguments passed to install_github

dots
: A list of ..., e.g., list(...). Only for internal use.

Details

parseGitHub turns the single character string representation into 3 or 4: Account, Repo, Branch, SubFolder.

Value

parseGitHub returns a data.table with added columns.

install_githubV returns a named character vector indicating packages successfully installed, unless the word "Failed" is returned, indicating installation failure. The names will be the full GitHub package name, as provided to gitPkgNames in the function call.

Examples

```
## Not run:
  install_githubV(c("PredictiveEcology/Require", "PredictiveEcology/quickPlot"))

## End(Not run)
```

---

pkgDep *Determine package dependencies*

Description

This will first look in local filesystem (in .libPaths()) and will use a local package to find its dependencies. If the package doesn't exist locally, including whether it is the correct version, then it will look in (currently) CRAN and its archives (if the current CRAN version is not the desired version to check). It will also look on GitHub if the package description is of the form of a GitHub package with format account/repo@branch or account/repo@commit. For this, it will attempt to get package dependencies from the GitHub 'DESCRIPTION' file. This is intended to replace tools::package_dependencies or pkgDep in the **miniCRAN** package, but with modifications to allow multiple sources to be searched in the same function call.

pkgDep2 is a convenience wrapper of pkgDep that "goes one level in", i.e., the first order dependencies, and runs the pkgDep on those.

This is a wrapper around tools::dependsOnPkgs, but with the added option of sorted, which will sort them such that the packages at the top will have the least number of dependencies that are in pkgs. This is essentially a topological sort, but it is done heuristically. This can be used to e.g., detach or unloadNamespace packages in order so that they each of their dependencies are detached or unloaded first.

pkgDepAlt is a newer, still experimental approach to pkgDep, which has different internal algorithms. With current testing, it appears to be slightly more accurate for (some unknown, as of yet)

edge cases. One known case is when the a package is installed locally package, but is not the version that is requested with pkgDep, the function will default to the local, installed, and incorrect package dependencies. pkgDepAlt gets this case correct. This function may eventually replace pkgDep.

Usage

```
pkgDep(
  packages,
  libPath = .libPaths(),
  which = c("Depends", "Imports", "LinkingTo"),
  recursive = FALSE,
  depends,
  imports,
  suggests,
  linkingTo,
  repos = getOption("repos"),
  keepVersionNumber = TRUE,
  includeBase = FALSE,
  sort = TRUE,
  purge = getOption("Require.purge", FALSE)
)

pkgDep2(
  packages,
  recursive = TRUE,
  which = c("Depends", "Imports", "LinkingTo"),
  depends,
  imports,
  suggests,
  linkingTo,
  repos = getOption("repos"),
  sorted = TRUE,
  purge = getOption("Require.purge", FALSE)
)

pkgDepTopoSort(
  pkgs,
  deps,
  reverse = FALSE,
  topoSort = TRUE,
  useAllInSearch = FALSE,
  returnFull = TRUE,
  recursive = TRUE,
  purge = getOption("Require.purge", FALSE)
)

pkgDepAlt(
  packages,
```

```
  libPath = .libPaths(),
  which = c("Depends", "Imports", "LinkingTo", "Remotes"),
  recursive = FALSE,
  depends,
  imports,
  suggests,
  linkingTo,
  enhances,
  remotes,
  repos = getOption("repos"),
  keepVersionNumber = TRUE,
  includeBase = FALSE,
  sort = TRUE,
  purge = getOption("Require.purge", FALSE)
)
```

Arguments

packages	Character vector of packages to install via `install.packages`, then load (i.e., with `library`). If it is one package, it can be unquoted (as in `require`). In the case of a GitHub package, it will be assumed that the name of the repository is the name of the package. If this is not the case, then pass a named character vector here, where the names are the package names that could be different than the GitHub repository name.
libPath	A path to search for installed packages. Defaults to `.libPaths()`
which	a character vector listing the types of dependencies, a subset of `c("Depends","Imports","LinkingTo",` Character string `"all"` is shorthand for that vector, character string `"most"` for the same vector without `"Enhances"`.
recursive	Logical. Should dependencies of dependencies be searched, recursively. NOTE: Dependencies of suggests will not be recursive. Default `TRUE`.
depends	Logical. Include packages listed in "Depends". Default `TRUE`.
imports	Logical. Include packages listed in "Imports". Default `TRUE`.
suggests	Logical. Include packages listed in "Suggests". Default `FALSE`.
linkingTo	Logical. Include packages listed in "LinkingTo". Default `TRUE`.
repos	The remote repository (e.g., a CRAN mirror), passed to either `install.packages`, `install_github` or `installVersions`.
keepVersionNumber	Logical. If `TRUE`, then the package dependencies returned will include version number. Default is `FALSE`
includeBase	Logical. Should R base packages be included, specifically, those in `tail(.libPath(),1)`
sort	Logical. If `TRUE`, the default, then the packages will be sorted alphabetically. If `FALSE`, the packages will not have a discernible order as they will be a concatenation of the possibly recursive package dependencies.
purge	Logical. Should all caches be purged Default is `getOption("Require.purge",FALSE)`. There is a lot of internal caching of results throughout the `Require` package.

	These help with speed and reduce calls to internet sources. However, sometimes these caches must be purged. The cached values are renewed when found to be too old, with the age limit. This maximum age can be set in seconds with the environment variable R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE, or if unset, defaults to 3600 (one hour – see available.packages). Internally, there are calls to available.packages
sorted	Logical. If TRUE, the default, the packages will be sorted in the returned list from most number of dependencies to least.
pkgs	A vector of package names to evaluate their reverse depends (i.e., the packages that *use* each of these packages)
deps	An optional named list of (reverse) dependencies. If not supplied, then tools::dependsOnPkgs(...,rec = TRUE) will be used
reverse	Logical. If TRUE, then this will use tools::pkgDependsOn to determine which packages depend on the pkgs
topoSort	Logical. If TRUE, the default, then the returned list of packages will be in order with the least number of dependencies listed in pkgs at the top of the list.
useAllInSearch	Logical. If TRUE, then all non-core R packages in search() will be appended to pkgs to allow those to also be identified
returnFull	Logical. Primarily useful when reverse = TRUE. If TRUE, then then all installed packages will be searched. If FALSE, the default, only packages that are currently in the search() path and passed in pkgs will be included in the possible reverse dependencies.
enhances	Logical. Include packages listed in "Enhances". Default FALSE.
remotes	Logical. Include packages listed in "Remotes". This is only relevant for GitHub packages. Default TRUE.

Value

A possibly ordered, named (with packages as names) list where list elements are either full reverse depends.

Note

tools::package_dependencies and pkgDep will differ under the following circumstances:

1. GitHub packages are not detected using tools::package_dependencies;
2. tools::package_dependencies does not detect the dependencies of base packages among themselves, *e.g.*, methods depends on stats and graphics.

Examples

```
## Not run:
  pkgDep("Require")
  pkgDep("Require", keepVersionNumber = FALSE) # just names
  pkgDep("PredictiveEcology/reproducible") # GitHub
  pkgDep("PredictiveEcology/reproducible", recursive = TRUE) # GitHub
 pkgDep(c("PredictiveEcology/reproducible", "Require")) # GitHub package and local packages
```

```
 pkgDep(c("PredictiveEcology/reproducible", "Require", "plyr")) # GitHub, local, and CRAN packages

## End(Not run)
## Not run:
  pkgDep2("Require")
  # much bigger one
  pkgDep2("reproducible")

## End(Not run)
## Not run:
pkgDepTopoSort(c("Require", "data.table"), reverse = TRUE)

## End(Not run)
```

---

pkgSnapshot *Take a snapshot of all the packages and version numbers*

Description

This can be used later by `installVersions` to install or re-install the correct versions.

Usage

```
pkgSnapshot(
  packageVersionFile = "packageVersions.txt",
  libPaths,
  standAlone = FALSE,
  purge = getOption("Require.purge", FALSE)
)
```

Arguments

packageVersionFile	A filename to save the packages and their currently installed version numbers. Defaults to `".packageVersions.txt"`.
libPaths	The path to the local library where packages are installed. Defaults to the `.libPaths()[1]`.
standAlone	Logical. If `TRUE`, all packages will be installed to and loaded from the `libPaths` only. If `FALSE`, then `libPath` will be prepended to `.libPaths()` during the `Require` call, resulting in shared packages, i.e., it will include the user's default package folder(s). This can be create dramatically faster installs if the user has a substantial number of the packages already in their personal library. Default `FALSE` to minimize package installing.
purge	Logical. Should all caches be purged Default is `getOption("Require.purge",FALSE)`. There is a lot of internal caching of results throughout the `Require` package. These help with speed and reduce calls to internet sources. However, sometimes these caches must be purged. The cached values are renewed when found to be

too old, with the age limit. This maximum age can be set in seconds with the environment variable R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE, or if unset, defaults to 3600 (one hour – see available.packages).

Internally, there are calls to available.packages

Details

A file is written with the package names and versions of all packages within libPaths. This can later be passed to Require.

Examples

```
pkgSnapFile <- tempfile()
pkgSnapshot(pkgSnapFile, .libPaths()[1])
data.table::fread(pkgSnapFile)

## Not run:

# An example to move this file to a new computer
library(Require)
setLibPaths(.libPaths()[1])  # this will only do a snapshot of the main user library
fileName <- "packageSnapshot.txt"
pkgSnapshot(fileName)
# Get file on another computer -- via email, slack, cloud, etc.
# library(googledrive)
# (out <- googledrive::drive_upload(fileName)) # copy the file id to clipboard

# On new machine
fileName <- "packageSnapshot.txt"
library(Require)
# get the file from email, slack, cloud etc.
# library(googledrive)
# drive_download(as_id(PASTE-THE-FILE-ID-HERE), path = fileName)
setLibPaths("~/RPackages") # start with an empty folder for new
                           # library to minimize package version conflicts
Require(packageVersionFile = fileName)

## End(Not run)
```

RequireOptions Require *options*

Description

These provide top-level, powerful settings for a comprehensive reproducible workflow. To see defaults, run RequireOptions(). See Details below.

Usage

```
RequireOptions()
```

Details

Below are options that can be set with `options("Require.xxx" = newValue)`, where `xxx` is one of the values below, and `newValue` is a new value to give the option. Sometimes these options can be placed in the user's `.Rprofile` file so they persist between sessions.

The following options are likely of interest to most users:

`RPackageCache` Default: `NULL`. If a folder is provided, then binary and source packages will be cached here. Subsequent downloads of same package will use local copy. Default is to have packages not be cached locally so each install of the same version will be from the original source, e.g., CRAN, GitHub.

`buildBinarieis` Default: `TRUE`. Only relevant on *nix systems and if `getOption("Require.RPackageCache")` is set to a path. If `TRUE`, then `Require` will pass `INSTALL_OPTS = "--build"`, meaning the package binary will be built and then saved in the `getOption("Require.RPackageCache")`. This means that subsequent installs of this package on this or identical system will be faster.

`persistentPkgEnv` Default: `FALSE`. (ADVANCED USE) `Require` stashes a lot of information in a hidden environment, located at `Require:::.pkgEnv`. This gets reset at each restart of R and each reload of Require. To make the stashes more persistent, set this option to `TRUE`. A file will be placed at `file.path("~","._Require_pkgEnv.rdata")`, which will be restored at package load

`purge` Default: `FALSE`. If set to (almost) all internal caches used by `Require` will be deleted and rebuilt. This should not generally be necessary as it will automatically be deleted after (by default) 1 hour (set via `R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE` environment variable in seconds)

`unloadNamespaces` Default: `TRUE`. (ADVANCED USE) `Require` will attempt to detach and unload packages that conflict with the requested package installing via `Require`. This can be complicated, resulting in broken states that can only be recovered by restarting R. Default is to attempt to do this. `FALSE` will not attempt to do this. User must deal with inability to install packages due to package already being loaded.

`verbose` Default: `0`. During a `Require`, there is a lot of information collected and used. With `verbose` set to `1` or `2`, more of this information will be reported as an attribute attached to the return object of `Require`. This may help diagnosing problems.

---

setLibPaths *Set* `.libPaths`

Description

This will set the `.libPaths()` by either adding a new path to it if `standAlone = FALSE`, or will concatenate `c(libPath,tail(.libPaths(),1))` if `standAlone = TRUE`.

Usage

```
setLibPaths(libPaths, standAlone = TRUE)
```

Arguments

libPaths	A new path to append to, or replace all existing user components of .libPath()
standAlone	Logical. If TRUE, all packages will be installed to and loaded from the libPaths only. If FALSE, then libPath will be prepended to .libPaths() during the Require call, resulting in shared packages, i.e., it will include the user's default package folder(s). This can be create dramatically faster installs if the user has a substantial number of the packages already in their personal library. Default FALSE to minimize package installing.

Details

This code was modified from https://github.com/milesmcbain. A different, likely non-approved by CRAN approach that also works is here: https://stackoverflow.com/a/36873741/3890027.

Value

The main point of this function is to set .libPaths(), which will be changed as a side effect of this function. As when setting options, this will return the previous state of .libPaths() allowing the user to reset easily.

Examples

```
## Not run:
orig <- setLibPaths("~/newProjectLib") # will only have 2 paths,
                                       # this and the last one in .libPaths()
.libPaths() # see the 2 paths
setLibPaths(orig) # reset
.libPaths() # see the 2 original paths back

# will have 2 or more paths
setLibPaths("~/newProjectLib", standAlone = FALSE) # will have 2 or more paths


## End(Not run)
```

---

tempdir2 *Make a temporary (sub-)directory*

Description

Create a temporary subdirectory in .RequireTempPath(), or a temporary file in that temporary subdirectory.

Usage

```
tempdir2(sub = "", tempdir = getOption("Require.tempPath", .RequireTempPath()))
```

Arguments

sub	Character string, length 1. Can be a result of `file.path("smth","smth2")` for nested temporary sub directories.
tempdir	Optional character string where the temporary dir should be placed. Defaults to `.RequireTempPath()`

See Also

tempfile2

---

tempfile2 *Make a temporary subfile in a temporary (sub-)directory*

Description

Make a temporary subfile in a temporary (sub-)directory

Usage

```
tempfile2(
  sub = "",
  tempdir = getOption("Require.tempPath", .RequireTempPath()),
  ...
)
```

Arguments

sub	Character string, length 1. Can be a result of `file.path("smth","smth2")` for nested temporary sub directories.
tempdir	Optional character string where the temporary dir should be placed. Defaults to `.RequireTempPath()`
...	passed to `tempfile`, e.g., `fileext`

See Also

tempdir2

`trimVersionNumber` *Trim version number off a compound package name*

Description

The resulting string(s) will have only name (including github.com repository if it exists).

Usage

```
trimVersionNumber(pkgs)
```

Arguments

`pkgs`	A character string vector of packages with or without GitHub path or versions

See Also

`extractPkgName`

Examples

```
trimVersionNumber("PredictiveEcology/Require (<=0.0.1)")
```

Index