

# Package ‘Require’

May 22, 2023

**Type** Package

**Title** Installing and Loading R Packages for Reproducible Workflows

**Description** A single key function, 'Require' that makes rerun-tolerant versions of 'install.packages' and `require` for CRAN packages, packages no longer on CRAN (i.e., archived), specific versions of packages, and GitHub packages. This approach is developed to create reproducible workflows that are flexible and fast enough to use while in development stages, while able to build snapshots once a stable package collection is found. 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, with subsequent times being sufficiently fast that they can be run every time without undue waiting burden on the user or developer.

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

**Date** 2023-05-22

**Version** 0.3.1

**Depends** R (>= 4.0)

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

**Suggests** covr, parallel, remotes, testit

**Encoding** UTF-8

**Language** en-CA

**License** GPL-3

**BugReports** <https://github.com/PredictiveEcology/Require/issues>

**ByteCompile** yes

**RoxygenNote** 7.2.3

**NeedsCompilation** no

**Author** Eliot J B McIntire [aut, cre] (<<https://orcid.org/0000-0002-6914-8316>>),  
Alex M Chubaty [ctb] (<<https://orcid.org/0000-0001-7146-8135>>),  
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** 2023-05-22 21:00:02 UTC

## R topics documented:

Require-package . . . . .	3
.downloadFileMasterMainAuth . . . . .	8
archiveVersionsAvailable . . . . .	9
availablePackagesOverride . . . . .	10
availableVersionOK . . . . .	11
checkPath . . . . .	11
clearRequirePackageCache . . . . .	12
DESCRIPTIONFileVersionV . . . . .	13
detachAll . . . . .	14
extractPkgName . . . . .	15
getOptionRPackageCache . . . . .	16
invertList . . . . .	16
linkOrCopy . . . . .	17
messageDF . . . . .	17
modifyList2 . . . . .	19
normPath . . . . .	20
paddedFloatToChar . . . . .	21
parseGitHub . . . . .	22
pkgDep . . . . .	23
pkgDepIfDepRemoved . . . . .	26
pkgSnapshot . . . . .	28
RequireCacheDir . . . . .	30
RequireOptions . . . . .	31
rversions . . . . .	32
setdiffNamed . . . . .	33
setLibPaths . . . . .	34
setLinuxBinaryRepo . . . . .	36
setup . . . . .	36
sourcePkgs . . . . .	37
tempdir2 . . . . .	38
tempfile2 . . . . .	38
trimVersionNumber . . . . .	39

**Index**

**40**

---

Require-package	<i>Require: Installing and Loading R Packages for Reproducible Workflows</i>
-----------------	--

---

## Description

A single key function, 'Require' that makes rerun-tolerant versions of 'install.packages' and 'require' for CRAN packages, packages no longer on CRAN (i.e., archived), specific versions of packages, and GitHub packages. This approach is developed to create reproducible workflows that are flexible and fast enough to use while in development stages, while able to build snapshots once a stable package collection is found. 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, with subsequent times being sufficiently fast that they can be run every time without undue waiting burden on the user or developer.

This is an "all in one" function that will run `install.packages` for CRAN and GitHub <https://github.com/> packages and will install specific versions of each package if versions are specified either via an (in)equality (e.g., "`glue (>=1.6.2)`" or "`glue (==1.6.2)`" 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(INSTALL_opts = "--no-multiarch"),  
  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),  
  type = getOption("pkgType"),  
  upgrade = FALSE,  
  ...  
)  
  
Install(  
  packages,  
  packageVersionFile,  
  libPaths,  
  install_githubArgs = list(),
```

```

install.packagesArgs = list(INSTALL_opts = "--no-multiarch"),
standAlone = getOption("Require.standAlone", FALSE),
install = TRUE,
repos = getOption("repos"),
purge = getOption("Require.purge", FALSE),
verbose = getOption("Require.verbose", FALSE),
type = getOption("pkgType"),
upgrade = FALSE,
...
)

```

## Arguments

<code>packages</code>	Character vector of packages to install via <code>install.packages</code> , then load (i.e., with <code>library</code> ). If it is one package, it can be unquoted (as in <code>require</code> ). 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.
<code>packageVersionFile</code>	Character string of a file name or logical. If TRUE, then this function will load the default file, <code>getOption("Require.packageVersionFile")</code> . If this argument is provided, then this will override all any packages passed to <code>packages</code> .
<code>libPaths</code>	The library path (or libraries) where all packages should be installed, and looked for to load (i.e., call <code>library</code> ). This can be used to create isolated, stand alone package installations, if used with <code>standAlone = TRUE</code> . Currently, the path supplied here will be prepended to <code>.libPaths()</code> (temporarily during this call) to <code>Require</code> if <code>standAlone = FALSE</code> or will set (temporarily) <code>.libPaths()</code> to <code>c(libPaths, tail(libPaths(), 1))</code> to keep base packages.
<code>install_githubArgs</code>	Deprecated. Values passed here are merged with <code>install.packagesArgs</code> , with the <code>install.packagesArgs</code> taking precedence if conflicting.
<code>install.packagesArgs</code>	List of optional named arguments, passed to <code>install.packages</code> . Default is only <code>--no-multi-arch</code> , meaning that only the current architecture will be built and installed (e.g., 64 bit, not 32 bit, in many cases).
<code>standAlone</code>	Logical. If TRUE, all packages will be installed to and loaded from the <code>libPaths</code> only. NOTE: If TRUE, THIS WILL CHANGE THE USER'S <code>.libPaths()</code> , similar to e.g., the <code>checkpoint</code> package. If FALSE, then <code>libPath</code> will be prepended to <code>.libPaths()</code> during the <code>Require</code> 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.
<code>install</code>	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., <code>install.packages</code> . If TRUE, the default, then this function will try to install any missing packages or dependencies.

require	Logical or character string. If TRUE, the default, then the function will attempt to call require on all requested packages, possibly after they are installed. If a character string, then it will only call require on those specific packages (i.e., it will install the ones listed in packages, but load the packages listed in require)
repos	The remote repository (e.g., a CRAN mirror), passed to either <code>install.packages</code> , <code>install_github</code> or <code>installVersions</code> .
purge	Logical. Should all caches be purged? Default is <code>getOption("Require.purge", FALSE)</code> . 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 <code>R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE</code> , or if unset, defaults to 3600 (one hour – see <a href="#">utils::available.packages</a> ). Internally, there are calls to <code>available.packages</code> .
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose $\geq 2$ , the return object will have an attribute: <code>attr(., "Require")</code> which has lots of information about the processes of the installs.
type	See <code>utils::install.packages</code>
upgrade	When FALSE, the default, will only upgrade a package when the version on in the local library is not adequate for the version requirements of the packages. Note: for convenience, <code>update</code> can be used for this argument.
...	Passed to <code>install.packages</code> . Good candidates are e.g., <code>type</code> or <code>dependencies</code> . This can be used with <code>install_githubArgs</code> or <code>install.packageArgs</code> which give individual options for those 2 internal function calls.

## Details

Install is the same as `Require(..., require = FALSE)`, for convenience.

## Value

Require is intended to replace `base::require`, thus it returns a logical, named vector indicating whether the named packages have been loaded. Because Require also has the ability to install packages, a return value of FALSE does not mean that it did not install correctly; rather, it means it did not attach with require, which could be because it did not install correctly, or also because e.g., `require = FALSE`.

`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 with `remotes::install_github`, it is not possible to specify a past version of a GitHub package unless that version is a tag or the user passes

the SHA that had that package version. Similarly, if a developer does a local install e.g., via `pkgload::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 `remotes::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 an R-version specific subfolder of `getOption("Require.RPackageCache")` whose default is `RPackageCache()`, meaning *cache packages locally in a project-independent location*, and will reuse them if needed. To turn off this feature, set `options("Require.RPackageCache" = FALSE)`.

### Note

For advanced use and diagnosis, the user can set `verbose = TRUE` or 1 or 2 (or via `options("Require.verbose")`). This will attach 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:

- Alex M Chubaty <achubaty@for-cast.ca> ([ORCID](#)) [contributor]
- 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
opts <- Require:::.setupExample()

library(Require)
getCRANrepos(ind = 1)
Require("stats") # analogous to require(stats), but it checks for
# pkg dependencies, and installs them, if missing

if (Require:::.runLongExamples()) {
  # Install in a new local library (libPaths)
  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)
  Install("crayon", libPaths = tempPkgFolder, standAlone = TRUE)

  # make a package version snapshot of installed packages
  tf <- tempfile()
  (pkgSnapshot(tf, standAlone = TRUE))

  # Change the libPaths to emulate a new computer or project
  tempPkgFolder <- file.path(tempdir(), "Packages2")
  # Reinstall and reload the exact version from previous
  Require(packageVersionFile = tf, libPaths = tempPkgFolder, standAlone = TRUE)

  # Mutual dependencies, only installs once -- e.g., curl
  tempPkgFolder <- file.path(tempdir(), "Packages")
  Install(c("remotes", "testit"), libPaths = tempPkgFolder, standAlone = TRUE)

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

  #####
  # Isolated projects -- Use a project folder and pass to libPaths or set .libPaths() #
  #####
  # GitHub packages
  ProjectPackageFolder <- file.path(tempdir(), "ProjectA")
  Require("PredictiveEcology/fpCompare@development",
    libPaths = ProjectPackageFolder, standAlone = FALSE
  )

  Install("PredictiveEcology/fpCompare@development",
    libPaths = ProjectPackageFolder,
    standAlone = TRUE
  ) # the latest version on GitHub

  #####
  # Mixing and matching GitHub, CRAN, with and without version numbering
  #####
  pkgs <- c(

```

```

    "remotes (<=2.4.1)", # old version
    "digest (>= 0.6.28)", # recent version
    "PredictiveEcology/fpCompare@a0260b8476b06628bba0ae73af3430cce9620ca0" # exact version
  )
  Require::Require(pkgs, libPaths = ProjectPackageFolder)
  Require:::cleanup(opts)
}

## End(Not run)

```

---

```

.downloadFileMasterMainAuth
      GITHUB_PAT-aware and main-master-aware download from
      GitHub

```

---

## Description

Equivalent to `utils::download.file`, but taking the `GITHUB_PAT` environment variable and using it to access the Github url.

## Usage

```

.downloadFileMasterMainAuth(
  url,
  destfile,
  need = "HEAD",
  verbose = getOption("Require.verbose"),
  verboseLevel = 2
)

```

## Arguments

<code>url</code>	a <a href="#">character</a> string (or longer vector for the "libcurl" method) naming the URL of a resource to be downloaded.
<code>destfile</code>	a character string (or vector, see the <code>url</code> argument) with the file path where the downloaded file is to be saved. Tilde-expansion is performed.
<code>need</code>	If specified, user can suggest which master or main or HEAD to try first. If unspecified, HEAD is used.
<code>verbose</code>	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in <code>Require</code> function, when <code>verbose &gt;= 2</code> , the return object will have an attribute: <code>attr(, "Require")</code> which has lots of information about the processes of the installs.
<code>verboseLevel</code>	A numeric indicating what verbose threshold (level) above which this message will show.



**Value**

This is called for its side effect, namely, the same as `utils::download.file`, but using a `GITHUB_PAT`, if it is in the environment, and trying both `master` and `main` if the actual url specifies either `master` or `main` and it does not exist.

---

archiveVersionsAvailable

*Available and archived versions*

---

**Description**

These are wrappers around `available.packages` and also get the archived versions available on CRAN.

**Usage**

```
archiveVersionsAvailable(package, repos)

available.packagesCached(
  repos,
  purge,
  verbose = getOption("Require.verbose"),
  returnDataTable = TRUE,
  type
)
```

**Arguments**

package	A single package name (without version or github specifications)
repos	The remote repository (e.g., a CRAN mirror), passed to either <code>install.packages</code> , <code>install_github</code> or <code>installVersions</code> .
purge	Logical. Should all caches be purged? Default is <code>getOption("Require.purge", FALSE)</code> . There is a lot of internal caching of results throughout the <code>Require</code> 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 <code>R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE</code> , or if unset, defaults to 3600 (one hour – see <a href="#">utils::available.packages</a> ). Internally, there are calls to <code>available.packages</code> .
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or <code>FALSE</code> , then minimal outputs; if 1 or <code>TRUE</code> , more outputs; 2 even more. NOTE: in <code>Require</code> function, when <code>verbose &gt;= 2</code> , the return object will have an attribute: <code>attr(, "Require")</code> which has lots of information about the processes of the installs.

returnDataTable      Logical. If TRUE, the default, then the return is a data.table. Otherwise, it is a matrix, as per available.packages

type                  See utils::install.packages

### Details

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

---

availablePackagesOverride

*Create a custom "available.packages" object*

---

### Description

This is the mechanism by which install.packages determines which packages should be installed from where. With this override, we can indicate arbitrary repos, Package, File for each individual package.

### Usage

```
availablePackagesOverride(toInstall, repos, purge, type = getOption("pkgType"))
```

### Arguments

toInstall            A pkgDT object

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 [utils::available.packages](#)).

Internally, there are calls to available.packages.

type                 See utils::install.packages

---

availableVersionOK	<i>Needs VersionOnRepos, versionSpec and inequality columns</i>
--------------------	---

---

**Description**

Needs VersionOnRepos, versionSpec and inequality columns

**Usage**

```
availableVersionOK(pkgDT)
```

**Arguments**

pkgDT	A pkgDT object
-------	----------------

---

checkPath	<i>Check directory path</i>
-----------	-----------------------------

---

**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(tmpdir(), "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) # clean up
```

---

clearRequirePackageCache

*Clear Require Cache elements*

---

**Description**

Clear Require Cache elements

**Usage**

```
clearRequirePackageCache(
  packages,
  ask = interactive(),
  Rversion = rversion(),
  clearCranCache = FALSE,
  verbose = getOption("Require.verbose")
)
```

**Arguments**

packages	Either missing or a character vector of package names (currently cannot specify version number) to remove from the local Require Cache.
ask	Logical. If TRUE, then it will ask user to confirm
Rversion	An R version (major dot minor, e.g., "4.2"). Defaults to current R version.
clearCranCache	Logical. If TRUE, then this will also clear the local crancache cache, which is only relevant if options(Require.useCranCache = TRUE), i.e., if Require is using the crancache cache also
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose >= 2, the return object will have an attribute: attr(., "Require") which has lots of information about the processes of the installs.

---

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),
  verbose = getOption("Require.verbose")
)
```

**Arguments**

file	A file path to a DESCRIPTION file
purge	Logical. Should all caches be purged? Default is <code>getOption("Require.purge", FALSE)</code> . 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 <code>R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE</code> , or if unset, defaults to 3600 (one hour – see <code>utils::available.packages</code> ). Internally, there are calls to <code>available.packages</code> .
other	Any other keyword in a DESCRIPTION file that precedes a ":". The rest of the line will be retrieved.
pkg	A character string with a GitHub package specification (c.f. remotes)
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when <code>verbose &gt;= 2</code> , the return object will have an attribute: <code>attr(., "Require")</code> which has lots of information about the processes of the installs.

**Details**

`getGitHubDESCRIPTION` retrieves the DESCRIPTION file from GitHub.com

---

detachAll	<i>Detach and unload all packages</i>
-----------	---------------------------------------

---

**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,
  verbose = getOption("Require.verbose")
)
```

**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.
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose >= 2, the return object will have an attribute: attr(., "Require") which has lots of information about the processes of the installs.

**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	<i>Extract info from package character strings</i>
----------------	--

---

**Description**

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

**Usage**

```
extractPkgName(pkgs, filenames)
```

```
extractVersionNumber(pkgs, filenames)
```

```
extractInequality(pkgs)
```

```
extractPkgGitHub(pkgs)
```

**Arguments**

pkgs	A character string vector of packages with or without GitHub path or versions
filenames	Can be supplied instead of pkgs if it is a filename e.g., a .tar.gz or .zip that was downloaded from CRAN.

**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")
```

---

getOptionRPackageCache

*Get the option for Require.RPackageCache*

---

**Description**

First checks if an environment variable `Require.RPackageCache` is set and defines a path. If not set, checks whether the options("Require.RPackageCache") is set. If a character string, then it returns that. If TRUE, then use `RequirePkgCacheDir()`. If FALSE then returns NULL.

**Usage**

```
getOptionRPackageCache()
```

---

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**

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

**Value**

A list with 2 levels deep, inverted from 1



**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
```

---

linkOrCopy

*Create link to file, falling back to making a copy if linking fails.*


---

**Description**

First try to create a hardlink to the file. If that fails, try a symbolic link (symlink) before falling back to copying the file. "File" here can mean a file or a directory.

**Usage**

```
linkOrCopy(from, to, allowSymlink = FALSE)
```

```
fileRenameOrMove(from, to)
```

**Arguments**

from, to            character vectors, containing file names or paths.

allowSymlink       Logical. If FALSE, the default, then it will try `file.link` first, then `file.copy`, omitting the `file.symlink` step

---

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.

This will only show a message if the value of `verbose` is greater than the `verboseLevel`. This is mostly useful for developers of code who want to give users of their code easy access to how verbose their code will be. A developer of a function will place this `messageVerbose` internally, setting the `verboseLevel` according to how advanced they may want the message to be. 1 is a reasonable default for standard use, 0 would be for "a very important message for all users", 2 or above would be increasing levels of details for e.g., advanced use. If a user sets to -1 with this numeric approach, they can avoid all messaging.

**Usage**

```

messageDF(df, round, verbose = getOption("Require.verbose"), verboseLevel = 1)

messageVerbose(..., verbose = getOption("Require.verbose"), verboseLevel = 1)

messageVerboseCounter(
  pre = "",
  post = "",
  verbose = getOption("Require.verbose"),
  verboseLevel = 1,
  counter = 1,
  total = 1,
  minCounter = 1
)

```

**Arguments**

df	A data.frame, data.table, matrix
round	An optional numeric to pass to round
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose >= 2, the return object will have an attribute: attr(., "Require") which has lots of information about the processes of the installs.
verboseLevel	A numeric indicating what verbose threshold (level) above which this message will show.
...	Passed to install.packages. 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.
pre	A single text string to paste before the counter
post	A single text string to paste after the counter
counter	An integer indicating which iteration is being done
total	An integer indicating the total number to be done.
minCounter	An integer indicating the minimum (i.e., starting value)

**Value**

Used for side effects, namely messaging that can be turned on or off with different numeric values of verboseLevel. A user sets the verboseLevel for a particular message.

---

`modifyList2``modifyList` for multiple lists

---

## Description

This calls `utils::modifyList` iteratively using `base::Reduce`, so it can handle >2 lists. The subsequent list elements that share a name will override previous list elements with that same name. It also will handle the case where any list is a NULL. Note: default `keep.null = TRUE`, which is different than `modifyList`

## Usage

```
modifyList2(..., keep.null = FALSE)
```

```
modifyList3(..., keep.null = TRUE)
```

## Arguments

<code>...</code>	One or more named lists.
<code>keep.null</code>	If TRUE, NULL elements in <code>val</code> become NULL elements in <code>x</code> . Otherwise, the corresponding element, if present, is deleted from <code>x</code> .

## Details

More or less a convenience around `Reduce(modifyList, list(...))`, with some checks, and the addition of `keep.null = TRUE` by default.

## Note

`modifyList3` retains the original behaviour of `modifyList2` (prior to Oct 2022); however, it cannot retain NULL values in lists.

## Examples

```
modifyList2(list(a = 1), list(a = 2, b = 2))
modifyList2(list(a = 1), NULL, list(a = 2, b = 2))
modifyList2(
  list(a = 1), list(x = NULL), list(a = 2, b = 2),
  list(a = 3, c = list(1:10))
)
```

---

normPath	<i>Normalize filepath</i>
----------	---------------------------

---

### 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()
```

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

### 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) # clean up

```

---

paddedFloatToChar      *Convert numeric to character with padding*

---

### Description

This will pad floating point numbers, right or left. For integers, either class integer or functionally integer (e.g., 1.0), it will not pad right of the decimal. For more specific control or to get exact padding right and left of decimal, try the `stringi` package. It will also not do any rounding. See examples.

### Usage

```
paddedFloatToChar(x, padL = ceiling(log10(x + 1)), padR = 3, pad = "0")
```

### Arguments

<code>x</code>	numeric. Number to be converted to character with padding
<code>padL</code>	numeric. Desired number of digits on left side of decimal. If not enough, <code>pad</code> will be used to pad.
<code>padR</code>	numeric. Desired number of digits on right side of decimal. If not enough, <code>pad</code> will be used to pad.
<code>pad</code>	character to use as padding ( <code>nchar(pad) == 1</code> must be TRUE). Currently, can be only <code>"0"</code> or <code>" "</code> (i.e., space).

### Value

Character string representing the filename.

### Author(s)

Eliot McIntire and Alex Chubaty

**Examples**

```

paddedFloatToChar(1.25)
paddedFloatToChar(1.25, padL = 3, padR = 5)
paddedFloatToChar(1.25, padL = 3, padR = 1) # no rounding, so keeps 2 right of decimal

```

---

parseGitHub

*Parse a github package specification*


---

**Description**

This converts a specification like PredictiveEcology/Require@development into separate columns, "Account", "Repo", "Branch", "GitSubFolder" (if there is one)

**Usage**

```
parseGitHub(pkgDT, verbose = getOption("Require.verbose"))
```

**Arguments**

pkgDT	A pkgDT data.table.
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose >= 2, the return object will have an attribute: attr(., "Require") which has lots of information about the processes of the installs.

**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.

---

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 does not 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 unload namespace packages in order so that they each of their dependencies are detached or unloaded first.

## 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),  
  verbose = getOption("Require.verbose"),  
  includeSelf = TRUE,  
  type = getOption("pkgType")  
)
```

```
pkgDep2(  
  packages,  
  recursive = TRUE,  
  which = c("Depends", "Imports", "LinkingTo"),
```

```

    depends,
    imports,
    suggests,
    linkingTo,
    repos = getOption("repos"),
    sorted = TRUE,
    purge = getOption("Require.purge", FALSE),
    includeSelf = TRUE,
    verbose = getOption("Require.verbose")
)

pkgDepTopoSort(
  pkgs,
  deps,
  reverse = FALSE,
  topoSort = TRUE,
  libPath = .libPaths(),
  useAllInSearch = FALSE,
  returnFull = TRUE,
  recursive = TRUE,
  purge = getOption("Require.purge", FALSE),
  which = c("Depends", "Imports", "LinkingTo"),
  type = getOption("pkgType"),
  verbose = getOption("Require.verbose")
)

```

### Arguments

packages	Character vector of packages to install via <code>install.packages</code> , then load (i.e., with <code>library</code> ). If it is one package, it can be unquoted (as in <code>require</code> ). 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 <code>.libPaths()</code>
which	a character vector listing the types of dependencies, a subset of <code>c("Depends", "Imports", "LinkingTo", "Suggests", "Enhances")</code> . 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 <code>install.packages</code> , <code>install_github</code> or <code>installVersions</code> .



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 <code>tail(.libPath(), 1)</code>
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 <code>getOption("Require.purge", FALSE)</code> . 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 <code>R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE</code> , or if unset, defaults to 3600 (one hour – see <a href="#">utils::available.packages</a> ). Internally, there are calls to <code>available.packages</code> .
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when <code>verbose &gt;= 2</code> , the return object will have an attribute: <code>attr(., "Require")</code> which has lots of information about the processes of the installs.
includeSelf	Logical. If TRUE, the default, then the dependencies will include the package itself in the returned list elements, otherwise, only the "dependencies"
type	See <code>utils::install.packages</code>
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 <i>use</i> each of these packages)
deps	An optional named list of (reverse) dependencies. If not supplied, then <code>tools::dependsOnPkgs(..., recursive = TRUE)</code> will be used
reverse	Logical. If TRUE, then this will use <code>tools::pkgDependsOn</code> 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 <code>search()</code> will be appended to pkgs to allow those to also be identified
returnFull	Logical. Primarily useful when <code>reverse = TRUE</code> . If TRUE, then then all installed packages will be searched. If FALSE, the default, only packages that are currently in the <code>search()</code> path and passed in pkgs will be included in the possible reverse dependencies.

### 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:
if (Require:::.runLongExamples()) {
  opts <- Require:::.setupExample()

  pkgDep("tidyverse", recursive = TRUE)

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

  Require:::.cleanup(opts)
}

## End(Not run)
## Not run:
if (Require:::.runLongExamples()) {
  opts <- Require:::.setupExample()

  pkgDep2("reproducible")
  # much bigger one
  pkgDep2("tidyverse")

  Require:::.cleanup(opts)
}

## End(Not run)
## Not run:
if (Require:::.runLongExamples()) {
  opts <- Require:::.setupExample()

  pkgDepTopoSort(c("Require", "data.table"), reverse = TRUE)

  Require:::.cleanup(opts)
}

## End(Not run)
```

**Description**

This is primarily for package developers. It allows the testing of what the recursive dependencies would be if a package was removed from the immediate dependencies.

**Usage**

```
pkgDepIfDepRemoved(
  pkg = character(),
  depsRemoved = character(),
  verbose = getOption()
)
```

**Arguments**

pkg	A package name to be testing the dependencies
depsRemoved	A vector of package names who are to be "removed" from the pkg immediate dependencies
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose >= 2, the return object will have an attribute: attr(., "Require") which has lots of information about the processes of the installs.

**Value**

A list with 3 named lists `Direct`, `Recursive` and `IfRemoved`. `Direct` will show the top level direct dependencies, either `Remaining` or `Removed`. `Recursive` will show the full recursive dependencies, either `Remaining` or `Removed`. `IfRemoved` returns all package dependencies that are removed for each top level dependency. If a top level dependency is not listed in this final list, then it means that it is also a recursive dependency elsewhere, so its removal has no effect.

**Examples**

```
## Not run:
if (Require:::runLongExamples()) {
  opts <- Require:::setupExample()

  pkgDepIfDepRemoved("reproducible", "data.table")

  Require:::cleanup(opts)
}

## End(Not run)
```

---

 pkgSnapshot

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


---

## Description

This can be used later by Require to install or re-install the correct versions. See examples.

## Usage

```
pkgSnapshot(
  packageVersionFile = getOption("Require.packageVersionFile"),
  libPaths = .libPaths(),
  standAlone = FALSE,
  purge = getOption("Require.purge", FALSE),
  exact = TRUE,
  includeBase = FALSE,
  verbose = getOption("Require.verbose")
)
```

```
pkgSnapshot2(
  packageVersionFile = getOption("Require.packageVersionFile"),
  libPaths,
  standAlone = FALSE,
  purge = getOption("Require.purge", FALSE),
  exact = TRUE,
  includeBase = FALSE,
  verbose = getOption("Require.verbose")
)
```

## Arguments

packageVersionFile	A filename to save the packages and their currently installed version numbers. Defaults to "packageVersions.txt". If this is specified to be NULL, the function will return the exact Require call needed to install all the packages at their current versions. This can be useful to add to a script to allow for reproducibility of a script.
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. NOTE: If TRUE, THIS WILL CHANGE THE USER'S .libPaths(), similar to e.g., the checkpoint package. 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 <code>getOption("Require.purge", FALSE)</code> . 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 <code>R_AVAILABLE_PACKAGES_CACHE_CONTROL_MAX_AGE</code> , or if unset, defaults to 3600 (one hour – see <code>utils::available.packages</code> ). Internally, there are calls to <code>available.packages</code> .
exact	Logical. If TRUE, the default, then for GitHub packages, it will install the exact SHA, rather than the head of the <code>account/repo@branch</code> . For CRAN packages, it will install the exact version. If FALSE, then GitHub packages will identify their branch if that had been specified upon installation, not a SHA. If the package had been installed with reference to a SHA, then it will return the SHA as it does not know what branch it came from. Similarly, CRAN packages will report their version and specify with a <code>&gt;=</code> , allowing a subsequent user to install with a minimum version number, as opposed to an exact version number.
includeBase	Logical. Should R base packages be included, specifically, those in <code>tail(.libPath(), 1)</code>
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when <code>verbose &gt;= 2</code> , the return object will have an attribute: <code>attr(., "Require")</code> which has lots of information about the processes of the installs.

## Details

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

`pkgSnapshot2` returns a vector of package names and versions, with no file output. See examples.

## Value

Will both write a file, and (invisibly) return a vector of packages with the version numbers. This vector can be used directly in `Require`, though it should likely be used with `require = FALSE` to prevent attaching all the packages.

## Examples

```
## Not run:
if (Require:::runLongExamples()) {
  opts <- Require:::setupExample()
  # install one archived version so that below does something interesting

  libForThisEx <- tempdir2("Example")
  Require("crayon (==1.5.1)", libPaths = libForThisEx, require = FALSE)
  # Normal use -- using the libForThisEx for example;
  # normally libPaths would be omitted to get all
  # packages in user or project library
  tf <- tempfile()
}
```

```

# writes to getOption("Require.packageVersionFile")
# within project; also returns a vector
# of packages with version
pkgs <- pkgSnapshot(
  packageVersionFile = tf,
  libPaths = libForThisEx
)

# Now move this file to another computer e.g. by committing in git,
# emailing, googledrive
# on next computer/project
Require(packageVersionFile = tf, libPaths = libForThisEx)

# Using pkgSnapshot2 to get the vector of packages and versions

tf <- tempfile()
pkgs <- pkgSnapshot2(
  packageVersionFile = tf,
  libPaths = libForThisEx
)
Require(pkgs, require = FALSE) # will install packages from previous line
# (likely want require = FALSE
# and not load them all)

Require:::cleanup(opts)
unlink(getOption("Require.packageVersionFile"))
}

## End(Not run)

```

---

RequireCacheDir	<i>Path to (package) cache directory</i>
-----------------	--

---

### Description

Sets (if create = TRUE) or gets the cache directory associated with the Require package.

### Usage

```
RequireCacheDir(create)
```

```
RequirePkgCacheDir(create)
```

### Arguments

create	A logical indicating whether the path should be created if it does not exist. Default is FALSE.
--------	---

**Details**

To set a different directory than the default, set the system variable: `R_USER_CACHE_DIR = "somePath"` and/or `R_REQUIRE_PKG_CACHE = "somePath"` e.g., in `.Renviron` file or `Sys.setenv()`. See Note below.

**Value**

If `!is.null(getOptionRPackageCache())`, i.e., a cache path exists, the cache directory will be created, with a README placed in the folder. Otherwise, this function will just return the path of what the cache directory would be.

**Note**

Currently, there are 2 different Cache directories used by Require: `RequireCacheDir` and `RequirePkgCacheDir`. The `RequirePkgCacheDir` is intended to be a sub-directory of the `RequireCacheDir`. If you set `Sys.setenv("R_USER_CACHE_DIR" = "somedir")`, then both the package cache and cache dirs will be set, with the package cache a sub-directory. You can, however, set them independently, if you set `"R_USER_CACHE_DIR"` and `"R_REQUIRE_PKG_CACHE"` environment variable. The package cache can also be set with `options("Require.RPackageCache" = "somedir")`.

---

 RequireOptions

 Require *options*


---

**Description**

These provide top-level, powerful settings for a comprehensive reproducible workflow. See Details below.

**Usage**

```
RequireOptions()
```

```
getRequireOptions()
```

**Details**

`RequireOptions()` prints the default values of package options set at startup, which may have been changed (e.g., by the user) during the current session.

`getRequireOptions()` prints the current values of package options.

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:

`install` Default: `TRUE`. This is the default argument to `Require`, but does not affect `Install`. If this is `FALSE`, then no installations will be attempted, and missing packages will result in an error.

- RPackageCache** Default: `getOptionRPackageCache()`, which must be either a path or a logical. To turn off package caching, set this to `FALSE`. This can be set using an environment variable e.g. `Sys.setenv(R_REQUIRE_PKG_CACHE = "somePath")`, or `Sys.setenv(R_REQUIRE_PKG_CACHE = "TRUE")`; if that is not set, then an either a path or logical option (`options(Require.RPackageCache = "somePath")` or `options(Require.RPackageCache = TRUE)`). If `TRUE`, the default folder location `RequirePkgCacheDir()` will be used. If this is `TRUE` or a path 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.
- otherPkgs** Default: A character vector of packages that are generally more successful if installed from Source on Unix-alikes. Since there are repositories that offer binary packages builds for Linux (e.g., RStudio Package Manager), the vector of package names indicated here will default to a standard CRAN repository, forcing a source install. See also `spatialPkgs` option, which does the same for spatial packages.
- 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)
- spatialPkgs** Default: A character vector of packages that are generally more successful if installed from Source on Unix-alikes. Since there are repositories that offer binary packages builds for Linux (e.g., RStudio Package Manager), the vector of package names indicated here will default to a standard CRAN repository, forcing a source install. See also `otherPkgs` option, which does the same for non-spatial packages.
- useCranCache** Default: `FALSE`. A user can optionally use the locally cached packages that are available due to a user's use of the `crancache` package.
- verbose** Default: 1. See `?Require`.

---

rversions

*R versions*


---

## Description

Reference table of R versions and their release dates (2018 and later).

## Usage

```
rversions
```

## Format

An object of class `data.frame` with 21 rows and 2 columns.



## Details

Update this as needed using `rversions::r_versions()`:

```
# install.packages("rversions")
v = rversions::r_versions()
keep = which(as.Date(v$date, format = "
              as.Date("2018-01-01", format = "
dput(v[keep, c("version", "date")])
```

---

setdiffNamed

*Like setdiff, but takes into account names*

---

## Description

This will identify the elements in `l1` that are not in `l2`. If `missingFill` is provided, then elements that are in `l2`, but not in `l1` will be returned, assigning `missingFill` to their values. This might be `NULL` or `""`, i.e., some sort of empty value. This function will work on named lists, named vectors and likely on other named classes.

## Usage

```
setdiffNamed(l1, l2, missingFill)
```

## Arguments

<code>l1</code>	A named list or named vector
<code>l2</code>	A named list or named vector (must be same class as <code>l1</code> )
<code>missingFill</code>	A value, such as <code>NULL</code> or <code>""</code> or <code>"missing"</code> that will be given to the elements returned, that are in <code>l2</code> , but not in <code>l1</code>

## Details

There are 3 types of differences that might occur with named elements: 1. a new named element, 2. an removed named element, and 3. a modified named element. This function captures all of these. In the case of unnamed elements, e.g., `setdiff`, the first two are not seen as differences, if the values are not different.

## Value

A vector or list of the elements in `l1` that are not in `l2`, and optionally the elements of `l2` that are not in `l1`, with values set to `missingFill`

---

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`. Currently, the default is to make this new `.libPaths()` "sticky", meaning it becomes associated with the current directory even through a restart of R. It does this by adding and/updating the `‘.Rprofile’` file in the current directory. If this current directory is a project, then the project will have the new `.libPaths()` associated with it, even through an R restart.

### Usage

```
setLibPaths(
  libPaths,
  standAlone = TRUE,
  updateRprofile = getOption("Require.updateRprofile", FALSE),
  exact = FALSE,
  verbose = getOption("Require.verbose")
)
```

### Arguments

<code>libPaths</code>	A new path to append to, or replace all existing user components of <code>.libPath()</code>
<code>standAlone</code>	Logical. If TRUE, all packages will be installed to and loaded from the <code>libPaths</code> only. NOTE: If TRUE, THIS WILL CHANGE THE USER'S <code>.libPaths()</code> , similar to e.g., the checkpoint package. If FALSE, then <code>libPath</code> will be prepended to <code>.libPaths()</code> during the Require call, resulting in shared packages, i.e., it will include the user's default package folder(s). This can 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.
<code>updateRprofile</code>	Logical or Character string. If TRUE, then this function will put several lines of code in the current directory's <code>.Rprofile</code> file setting up the package libraries for this and future sessions. If a character string, then this should be the path to an <code>.Rprofile</code> file. To reset back to normal, run <code>setLibPaths()</code> without a <code>libPath</code> . Default: <code>getOption("Require.updateRprofile", FALSE)</code> , meaning FALSE, but it can be set with an option or within a single call.
<code>exact</code>	Logical. This function will automatically append the R version number to the <code>libPaths</code> to maintain separate R package libraries for each R version on the system. There are some cases where this behaviour is not desirable. Set <code>exact</code> to TRUE to override this automatic appending and use the exact, unaltered <code>libPaths</code> . Default is FALSE
<code>verbose</code>	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose

$\geq 2$ , the return object will have an attribute: `attr(., "Require")` which has lots of information about the processes of the installs.

## Details

This details of this code were 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:
if (Require:::.runLongExamples()) {
  opts <- Require:::.setupExample()
  origDir <- setwd(tempdir())
  td <- tempdir()
  setLibPaths(td) # set a new R package library locally
  setLibPaths() # reset it to original
  setwd(origDir)
  # Using standAlone = FALSE means that newly installed packages
  # will be installed
  # in the new package library, but loading packages can come
  # from any of the ones listed in .libPaths()

  # will have 2 or more paths
  otherLib <- file.path(td, "newProjectLib")
  setLibPaths(otherLib, standAlone = FALSE)
  # Can restart R, and changes will stay

  # remove the custom .libPaths()
  setLibPaths() # reset to previous; remove from .Rprofile
  # because libPath arg is empty

  Require:::.cleanup(opts)
  unlink(otherLib, recursive = TRUE)
}

## End(Not run)
```

---

setLinuxBinaryRepo      *Setup for binary Linux repositories*

---

### Description

Enable use of binary package builds for Linux from the RStudio Package Manager repo. This will set the repos option, affecting the current R session. It will put this binaryLinux in the first position. If the `getOption("repos")` is NULL, it will put backupCRAN in second position.

### Usage

```
setLinuxBinaryRepo(
  binaryLinux = "https://packagemanager.posit.co/",
  backupCRAN = srcPackageURLonCRAN
)
```

### Arguments

binaryLinux      A CRAN repository serving binary Linux packages.  
 backupCRAN      If there is no CRAN repository set

---

setup                      *Setup a project library, cache, options*

---

### Description

setup and setupOff are currently deprecated. These may be re-created in a future version. In its place, a user can simply put `.libPaths(libs, include.site = FALSE)` in their `.Rprofile` file, where `libs` is the directory where the packages should be installed and should be a folder with the R version number, e.g., derived by using `checkLibPaths(libs)`.

### Usage

```
setup(
  newLibPaths,
  RPackageFolders,
  RPackageCache = getOptionRPackageCache(),
  standAlone = getOption("Require.standAlone", TRUE),
  verbose = getOption("Require.verbose")
)

setupOff(removePackages = FALSE, verbose = getOption("Require.verbose"))
```

**Arguments**

newLibPaths	Same as RPackageFolders. This is for more consistent naming with Require(..., libPaths = ...).
RPackageFolders	One or more folders where R packages are installed to and loaded from. In the case of more than one folder provided, installation will only happen in the first one.
RPackageCache	See ?RequireOptions.
standAlone	Logical. If TRUE, all packages will be installed to and loaded from the libPaths only. NOTE: If TRUE, THIS WILL CHANGE THE USER'S .libPaths(), similar to e.g., the checkpoint package. 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.
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in Require function, when verbose >= 2, the return object will have an attribute: attr(.., "Require") which has lots of information about the processes of the installs.
removePackages	Deprecated. Please remove packages manually from the .libPaths()

---

sourcePkgs	<i>A list of R packages that should likely be installed from Source, not Binary</i>
------------	---

---

**Description**

The list of R packages that Require installs from source on Linux, even if the getOptions("repos") is a binary repository. This list can be updated by the user by modifying the options Require.spatialPkgs or Require.otherPkgs. Default "force source only packages" are visible with RequireOptions().

**Usage**

```
sourcePkgs(additional = NULL, spatialPkgs = NULL, otherPkgs = NULL)
```

**Arguments**

additional	Any other packages to be added to the other 2 argument vectors
spatialPkgs	A character vector of package names that focus on spatial analyses.
otherPkgs	A character vector of package names that often require system specific compilation.

**Value**

A sorted concatenation of the 3 input parameters.

---

tempdir2	<i>Make a temporary (sub-)directory</i>
----------	---

---

**Description**

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

**Usage**

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

**Arguments**

sub	Character string, length 1. Can be a result of <code>file.path("smth", "smth2")</code> for nested temporary sub directories.
tempdir	Optional character string where the temporary dir should be placed. Defaults to <code>.RequireTempPath()</code>
create	Logical. Should the directory be created. Default TRUE

**See Also**

[tempfile2\(\)](#)

---

tempfile2	<i>Make a temporary subfile in a temporary (sub-)directory</i>
-----------	--

---

**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 <code>file.path("smth", "smth2")</code> for nested temporary sub directories.
tempdir	Optional character string where the temporary dir should be placed. Defaults to <code>.RequireTempPath()</code>
...	passed to <code>tempfile</code> , e.g., <code>fileext</code>

**See Also**[tempdir2\(\)](#)

---

trimVersionNumber	<i>Trim version number off a compound package name</i>
-------------------	--

---

**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

- \* **datasets**
  - rversions, [32](#)
  - .downloadFileMasterMainAuth, [8](#)
- archiveVersionsAvailable, [9](#)
- available.packagesCached
  - (archiveVersionsAvailable), [9](#)
- availablePackagesOverride, [10](#)
- availableVersionOK, [11](#)
- base::Reduce, [19](#)
- character, [8](#)
- checkPath, [11](#)
- checkPath,character,logical-method
  - (checkPath), [11](#)
- checkPath,character,missing-method
  - (checkPath), [11](#)
- checkPath,missing,ANY-method
  - (checkPath), [11](#)
- checkPath,NULL,ANY-method (checkPath),  
[11](#)
- clearRequirePackageCache, [12](#)
- DESCRIPTIONFileOtherV
  - (DESCRIPTIONFileVersionV), [13](#)
- DESCRIPTIONFilePath, [13](#)
- detachAll, [14](#)
- dir.create(), [12](#)
- extractInequality (extractPkgName), [15](#)
- extractPkgGitHub (extractPkgName), [15](#)
- extractPkgName, [15](#)
- extractPkgName(), [39](#)
- extractVersionNumber (extractPkgName),  
[15](#)
- file.exists(), [12](#)
- fileRenameOrMove (linkOrCopy), [17](#)
- getGitHubDESCRIPTION
  - (DESCRIPTIONFileVersionV), [13](#)
- getOptionRPackageCache, [16](#)
- getRequireOptions (RequireOptions), [31](#)
- Install (Require-package), [3](#)
- invertList, [16](#)
- linkOrCopy, [17](#)
- messageDF, [17](#)
- messageVerbose (messageDF), [17](#)
- messageVerboseCounter (messageDF), [17](#)
- modifyList2, [19](#)
- modifyList3 (modifyList2), [19](#)
- normPath, [20](#)
- normPath,character-method (normPath), [20](#)
- normPath,list-method (normPath), [20](#)
- normPath,logical-method (normPath), [20](#)
- normPath,missing-method (normPath), [20](#)
- normPath,NULL-method (normPath), [20](#)
- paddedFloatToChar, [21](#)
- parseGitHub, [22](#)
- pkgDep, [23](#)
- pkgDep2 (pkgDep), [23](#)
- pkgDepIfDepRemoved, [26](#)
- pkgDepTopoSort (pkgDep), [23](#)
- pkgSnapshot, [28](#)
- pkgSnapshot2 (pkgSnapshot), [28](#)
- Require (Require-package), [3](#)
- Require-package, [3](#)
- RequireCacheDir, [30](#)
- RequireOptions, [31](#)
- RequirePkgCacheDir (RequireCacheDir), [30](#)
- rversions, [32](#)
- setdiffNamed, [33](#)
- setLibPaths, [34](#)
- setLinuxBinaryRepo, [36](#)
- setup, [36](#)



setupOff (setup), [36](#)  
sourcePkgs, [37](#)

tempdir2, [38](#)  
tempdir2(), [39](#)  
tempfile2, [38](#)  
tempfile2(), [38](#)  
trimVersionNumber, [39](#)  
trimVersionNumber(), [16](#)

utils::available.packages, [5](#), [9](#), [10](#), [14](#),  
[25](#), [29](#)  
utils::modifyList, [19](#)