# Package ‘multiversion’

March 21, 2022

**Type** Package

**Title** Version Controlled Package Manager

**Version** 0.3.6

**Description** Supports installing of multiple versions of a package and loading of specific versions and their dependencies. The only package that you need to install is this package providing the support for the complete library management. Migrate your ordinary library today!!

**License** LGPL-2

**Encoding** UTF-8

**RoxygenNote** 7.1.2

**Depends** R (>= 3.5.0)

**Suggests** testthat (>= 3.0.0), waldo, devtools, withr, callr

**Config/testthat/edition** 3

**NeedsCompilation** no

**Author** Siete C. Frouws [aut, cre]

**Maintainer** Siete C. Frouws <scfrouws@gmail.com>

**Repository** CRAN

**Date/Publication** 2022-03-21 18:50:05 UTC

## R topics documented:

---

`.set_test_lib_location` *Set lib.location to test_library*

---

## Description

Set lib.location to test_library

## Usage

```
.set_test_lib_location()
```

`bareVersion` *Remove ‘>‘ or ‘>=‘ from version string.*

## Description

Remove ‘>‘ or ‘>=‘ from version string.

## Usage

```
bareVersion(packVersion)
```

## Arguments

| | |
|---|---|
| `packVersion` | A version indication you would like to remove ‘>‘ and ‘>=‘ from. |

`chooseVersion` *Choose version based on the version indication, and available versions.*

## Description

Obtains the correct version based on the version instruction provided (e.g. >= `0.5`), the package name and it’s available versions. If no compatible version is found between the available versions a suitable error is thrown. All different version indications should be handled in this function, including:

1. a version with > or >= indicator.
2. just a version e.g. `'0.5.0'` (most specific)
3. a zero length char e.g. `''`

## Usage

```
chooseVersion(
  packVersion,
  versionList,
  packageName = "",
  pick.last = FALSE,
  warn_for_major_diff = TRUE
)
```

## Arguments

| | |
|---|---|
| `packVersion` | A single named version value. i.e. A package name and it's version requirement like: `c(dplyr = '>= 0.4.0')`. |
| `versionList` | A list of available versions for this package to choose from. It is the list to choose from and check availability. Created with `lib.available_versions`. |
| `packageName` | It is used for clear error handling. It should be the package name it is trying to load so we can mention it when crashing. |
| `pick.last` | See details. |
| `warn_for_major_diff` | If true, it will throw a warning when the requested package is a major release higher than that is specified. |

## Details

Note that both (1) and (3) are effected by 'pick.last'.

If a version like `>= 0.5` is given and multiple versions exist, a choice needs to be made. By default it will take the same or first higher version (when it exists, just `0.5` in the example). This most likely leads to not changing the behaviour of the code. Alternatively, picking the latest version is most likely to be accepted by other packages their dependencies (e.g. if a package that is loaded in the future depends on this package but asks for `> 0.6`, it will likely fail). The downside of this is that an update could be a major one, going from `0.5` to `2.0`, where allot of things can have changed and your code that used to work fine is at risk.

---

## `clean_download_catch` *Clean package download catch*

---

## Description

Clean the catch folder 'downloaded_packages' which lives in the temporary R session folder 'tempdir()'.

## Usage

```
clean_download_catch()
```

## Value

No return value, is called for it's side-effect of removing the `file.path(tempdir(),'downloaded_packages')` folder.

## `detachAll` *Detach all loaded packages and namespaces.*

### Description

Tries to detach all loaded packages and namespaces. Not always stable (within Rstudio). A restart of Rstudio might be required since it will often hold on to certain namespaces. A proper reset of all libraries is not possible, this is the best we can do.

In general, it is possible to create a complete clean environment by clearing your work space, running `detachAll` and then restarting Rstudio. If problems with package loading still persists, then follow the final alternative solution described in the details section of the documentation of `lib.load`.

### Usage

```
detachAll(reload_multiversion = FALSE, packageList = "all", dry_run = FALSE)
```

### Arguments

| | |
|---|---|
| `reload_multiversion` | If multiversion needs to be loaded again after everything (or all mentioned in `packageList`) is unloaded. |
| `packageList` | A character vector with the packages to detach/unload. Defaults to all packages (`names(sessionInfo()$otherPkgs`). When package X depends on package Y, make sure you first specify Y then X. |
| `dry_run` | If TRUE, lists all packages that will be cleaned up. |

### Value

When dry_run is FALSE, will returns the list of packages that it tried to detach. When not requested, will return them invisibly. In general, this function is called for it's side effect to unload all or some loaded packages.

## `detachIfExisting` *Detach package if it exists.*

### Description

Detach package if it exists.

### Usage

```
detachIfExisting(packageNames)
```

### Arguments

| | |
|---|---|
| `packageNames` | A vector of package names which need to be detached. Silently ignores when the package is not loaded. |

### Value

No return value, this function is called for it's side-effect. Detaches a package if it can find it. Will also try to unload package DLLs that might be loaded using `library.dynam.unload`.

---

## `error_packageAlreadyLoaded` *Throw error because this package is already loaded and not compatible with the requested version.*

---

### Description

Throw error because this package is already loaded and not compatible with the requested version.

### Usage

```
error_packageAlreadyLoaded(
  requested_package_name,
  requested_version,
  already_loaded_version
)
```

### Arguments

| | |
|---|---|
| `requested_package_name` | A single package name of the package that was desired to be loaded. |
| `requested_version` | The version definition (like: ">= 3.2.1") of the package that was tried to be loaded. |
| `already_loaded_version` | The version of the package that is already loaded, which causes this error to be fired. |

`lib.available_versions` *Just checks the multiversion library for all available versions installed for a specific package. If no name is provided, an error is returned.*

## Description

Just checks the multiversion library for all available versions installed for a specific package. If no name is provided, an error is returned.

## Usage

```
lib.available_versions(packageName, lib_location = lib.location())
```

## Arguments

| | |
|---|---|
| `packageName` | The name of the package for which all versions must be returned. |
| `lib_location` | The folder containing the structure where this package his versions need to be checked. |

## Value

A character vector with the different versions that are available for a specific package.

`lib.check_compatibility` *check if version indication is compliant.*

## Description

Returns TRUE if the 'condition' complies with the provided 'version'. This function is vectorized.

## Usage

```
lib.check_compatibility(condition, version)
```

## Arguments

| | |
|---|---|
| `condition` | A version indication like '>= 4.5.1' or '2.3.4' or '> 1.2.3' or '"' (empty) or 'NA'. |
| `version` | A version number like '1.2.3', or a vector of version strings (will be converted to 'numeric_version('1.2.3') during comparison). |

## Value

A logical indicating if the version is considered compatible.

## `lib.clean` *Clean multiversion library, revert to state of last commit.*

### Description

Clean up all un-tracked (not committed) installed libraries in the multiversion library. Will additionally also clean up the TEMP_install_location directory (this is an 'ignored' directory).

### Usage

```
lib.clean(lib_location = lib.location(), clean_temp_lib = TRUE)
```

### Arguments

| | |
|---|---|
| `lib_location` | By default the library path returned by `lib.location()` is used. See Note. |
| `clean_temp_lib` | If true, will also run `lib.clean_install_dir()`. |

### Details

Since it involves a quite invasive operation, it asks for permission when being called in an interactive session.

### Value

Clean up all un-tracked (not committed) installed libraries in the multiversion library. Will additionally also clean up the TEMP_install_location directory (this is an 'ignored' directory). Will ask for permission when being called in an interactive session.

### Note

It will build the most likely installation directory based on the `lib_location` you provide. See `lib.location_install_dir`. Which is `<your lib>/TEMP_install_location`.

## `lib.clean_install_dir` *Clear the temp install folder.*

### Description

The temporary installation folder (indicated by `lib.location_install_dir()`) is used to install the package before moving ('converting') it to the final location. This function removes this temporary folder. Make sure that all installed packages that are desired to keep are converted. You can run the `lib.convert()` once again to make sure this is the case.

### Usage

```
lib.clean_install_dir(
  lib_location = lib.location(),
  temp_install_location = lib.location_install_dir(lib_location)
)
```

### Arguments

| | |
|---|---|
| `lib_location` | By default the library path returned by `lib.location()` is used. It is only used to build the temp_install.location when that argument is not provided. |
| `temp_install_location` | The folder that is emptied by this function. |

### Value

No return value, it is called for it's side-effect of removing the temporary installation folder (located in `<multiversion_lib>/TEMP_install_location`). This must be called after every installation.

---

## `lib.clean_libPaths` *Exclude not relevant search paths.*

---

### Description

Excludes all `.libPaths` other then those needed for lib.load().

### Usage

```
lib.clean_libPaths(lib_location = lib.location(), dry_run = FALSE)
```

### Arguments

| | |
|---|---|
| `lib_location` | The folder which contains the multiversion library. All directories in `.libPaths()` containing this path will be kept. By default, it checks the environment variable `R_MV_LIBRARY_LOCATION` to find this directory. |
| `dry_run` | If TRUE, will not change the paths but will print the paths that would be removed by cleaning up the `.libPaths()` list. |

### Value

No return value, called for it's side effect of cleaning the `.libPaths` by removing any non-multiversion library locations.

`lib.convert` *Move normally installed packages to R_MV_library structure.*

## Description

After this conversion is completed and you configure (temporarily by using `lib.location(...)` or for eternity by setting the equally named environment variable) the R_MV_LIBRARY_LOCATION env var, you are good to go! You can directly use `lib.load` for loading packages. Thanks for using `multiversion`!!

This function creates the R_MV_library structure by moving normally installed packages to a parallel library structure. `<lib1>/BH/DESCRIPTION` becomes`<lib2>/BH/1.60.0-2/BH/DESCRIPTION` so that also `1.60.0-3` etc. can be installed.
This functionality is also used (with it's default values) for converting installed packages from the temporary installation directory to the final R_MV_library. The TEMP installation directory is in a standard flat library structure.

Note that it is really no problem to perform a conversion again, it will only move new versions of already present packages and will never overwrite. To continue with a clean Temp folder, run `lib.clean_install_dir()` which will remove the folder.

## Usage

```
lib.convert(
  source_lib = lib.location_install_dir(destination_mv_lib),
  destination_mv_lib = lib.location(),
  force_overwrite = FALSE,
  packages_to_convert
)
```

## Arguments

| | |
|---|---|
| `source_lib` | The temporary library where a package is temporarily installed (having a normal library structure). By default, the path is generated using `lib.location_install_dir()` on the `destination_mv_lib` that is provided which appends `/TEMP_install_location`. |
| `destination_mv_lib` | The folder containing a structure where all packages in the temp folder must be moved to. By default, it checks the environment variable `R_MV_LIBRARY_LOCATION` for this directory. |
| `force_overwrite` | If you are experimenting and you would like to overwrite the newly installed package. Normally only desired when the package you are experimenting with is a self maintained package and you are sure you increased the version to a new one. |
| `packages_to_convert` | A character vector with the names of the packages that need to be converted to the R_MV_library. If missing or empty, all will be converted. |

## Value

No return value, it is called for it's side-effect. Will convert a set of packages from a normal package library structure to a multiversion library version. By default, from the temporary multiversion installation directory to the final multiversion library.

## Examples

```
# As an experiment (or when getting started) you could run this with
# your complete standard library (not your base library).

#> lib.convert(source_lib = Sys.getenv("R_LIBS_USER"),
#>             destination_mv_lib  = "./REMOVE_ME_example_library")

# Running the same operation a second time will result
# in a notification that all files were already copied.

# Just running it will use the R_MV_library defined by the environment
# variable and look inside for the Temp folder to use.

#> lib.convert()

# It is sufficient to only provide the destination_mv_lib,
# it will look for the "/TEMP_install_location" folder as the 'source_lib' by default.

#> lib.convert(destination_mv_lib = "./R_MV_library")
```

---

`lib.decide_version` *Choose correct package version, and print decision.*

---

## Description

Obtains the correct version based on the version instruction provided (e.g. `>= 0.5`). It will print which version is chosen if ‘verbose = TRUE‘. if no compatible version is found between the available versions, the function ’chooseVersion’ will return an error to notify you.

## Usage

```
lib.decide_version(
  packVersion,
  lib_location,
  pick.last = FALSE,
  print_version_choice = TRUE,
  warn_for_major_diff = TRUE
)
```

### Arguments

| | |
|---|---|
| `packVersion` | A named character vector with package names and their version indication (e.g. `c(dplyr = '>= 0.4.0',ggplot = '')`). |
| `lib_location` | The location of the R_MV_library folder. |
| `pick.last` | If a version like>= `0.5` is given and multiple versions exist, a choice needs to be made. By default it will take the first higher version (when it exists, just`0.5`, which is often the case). This because this is most likely to not change the behavior of the code. Picking the latest version is most compatible with matching other packages their dependencies (e.g. if a later package depends on this package but asks for> `0.6`, it will crash). The downside of this is that an update could be a major one, going from`0.5` to`2.0`, where allot of things can change and code is likely to not work anymore. |
| `print_version_choice` | if true, it will print the choices it made. |
| `warn_for_major_diff` | If true, the default, will return warnings if the loaded package is a major release higher then the package that was requested. |

## `lib.dependencies` *List the dependencies of a package.*

### Description

Provide a package name (can be without quotes) to show its dependencies. To list all dependencies of the complete library, use the inverse function "`lib.dependsOnMe(all)`" with the value 'all'. That function also does not require quotes when calling it. So `lib.dependencies(package.a)` will work.

### Usage

```
lib.dependencies(
  packageName,
  do_print = TRUE,
  character.only = FALSE,
  lib_location = lib.location()
)
```

### Arguments

| | |
|---|---|
| `packageName` | The (unquoted) package name for which you would like to print the dependencies. |
| `do_print` | If true (default), prints the dependencies. In both cases, the dependencies are returned invisibly. |
| `character.only` | If TRUE, (FALSE by default), the package names can be provided as character vector. Otherwise, direct unquoted package names are supported. |

| | |
|---|---|
| `lib_location` | The folder containing the R_MV_library structure where this function observes the dependencies. By default, it checks the environment variable `R_MV_LIBRARY_LOCATION` for this directory. |

## Value

When do_print is TRUE, will print use `message` to show the provided package(s) his dependencies. Also returns the dependencies invisibly.

## Examples

```
## Not run:
    lib.dependencies(dplyr)
    lib.dependencies('devtools', character.only = TRUE)
    devtools_deps <- lib.dependencies(devtools, do_print = FALSE)

## End(Not run)
```

---

`lib.dependencies_online`
*Check a package his online dependencies*

---

## Description

Returns a `c(name = '<version spec>')` array which can be used for `lib.load()`, `lib.install_if_not_compatible()` or `lib.dependsOnMe()`.

## Usage

```
lib.dependencies_online(packageName, cran_url = "https://cran.rstudio.com/")
```

## Arguments

| | |
|---|---|
| `packageName` | The package name to check. |
| `cran_url` | Defaults to 'https://cran.rstudio.com/'. |

## Value

Returns a named character with the packages and their version conditions which the given package depends on.

## lib.dependsOnMe *Shows the dependencies of (all or) a certain function(s).*

### Description

Can be called without using quotes like `lib.dependsOnMe(dplyr)`. It supports the special feature `lib.dependsOnMe(all)`, which will print a list of all packages available with their dependencies.

A simple wrapper "`lib.installed_packages`", will do precisely that.

### Usage

```
lib.dependsOnMe(
  ...,
  checkMyDeps = NULL,
  lib_location = lib.location(),
  dont_print = FALSE
)
```

### Arguments

| | |
|---|---|
| `...` | All packages and their versions you would like to check e.g. `lib.dependsOnMe(DBI = '0.5',assertthat,R6 = '0.6',quietly = TRUE)`. |
| `checkMyDeps` | Supports providing a named character vector of packages and their versions instead of the direct input. Use it like this when calling it via another function. |
| `lib_location` | The folder containing a structure where this function observe the dependencies from. By default, it checks the environment variable `R_MV_LIBRARY_LOCATION` for this directory. |
| `dont_print` | When true, will not print anything, but will expect you to make use of the invisibly returned package character vector. |

### Value

It returns a special character array with package:version names for every package that has a dependency on the provided `checkMyDeps` or `...` condition.

## lib.devtools_load *Loads ‘devtools‘ version 1.13.1 and it’s dependencies.*

### Description

During the library call, `appendLibPaths` is TRUE, making sure that some devtools functionality (like running tests) in child R instances will still work and know where to load their libraries from.

### Usage

```
lib.devtools_load(lib_location = lib.location())
```

### Arguments

| | |
|---|---|
| `lib_location` | The (version controlled) library to load devtools from. Use `lib.install('devtools',allow_overwrit = TRUE)` to install devtools, if you have not done so already. |

### Value

No return value, called for it's side-effect of loading the devtools and testthat packages. Also the library paths of both packages will be added to the `.libPaths()`

---

`lib.execute_using_packagelist`

*Perform operation with a certain set of packages.*

---

### Description

This function can be used to perform R operations with a configured set of packages to load in a separate R process. The package `callr` is required to use this functionality. It will start a new process, then load the provided packages and execute your function. The `callr` package may be provided via the R_MV_library or your standard library, in which case it must be in a library where `.libPaths` is pointing to.

### Usage

```
lib.execute_using_packagelist(
  packages_to_load = c(),
  func_handle,
  ...,
  .lib_location = lib.location(),
  .pick_last = FALSE,
  .also_load_from_temp_lib = FALSE,
  .wait_for_response = TRUE,
  .run_quietly = FALSE,
  .callr_arguments = list()
)
```

### Arguments

| | |
|---|---|
| `packages_to_load` | An array indicating which packages must be loaded like "`c(dplyr = '0.5.0',ggplot2 = '',tidyr = '> 1.2.3')`". |
| `func_handle` | A function object or the function name as a character string. |

| | |
|---|---|
| `...` | Provide all the remaining arguments which will be arguments for the function handle. Note that every argument must be named and must match an argument in your func_handle. |
| `.lib_location` | The location of the version controlled library. Defaults to lib.location(), which is the directory provided by the environment variable. |
| `.pick_last` | Passed to `lib.load(packages_to_load,...)` inside the fired `callr` process. |
| `.also_load_from_temp_lib` | Passed to `lib.load(packages_to_load,...)` inside the fired `callr` process. |
| `.wait_for_response` | If false, it will fire and forget and return immediately using `callr::r_process`, otherwise will use `callr::r`. |
| `.run_quietly` | Controls the 'show' parameter of `callr::r` or `callr::r_process`. |
| `.callr_arguments` | List specifying additional arguments for `callr::r` or `callr::r_process` (depending on the `.wait_for_response` value). Note that `func`, `args`, `show` and `libpath` are already in use. Every parameter must be named. |

## Details

The additional arguments to callr: `.callr_arguments`, can for example be used to keep a log of a detached process. By including the following `.callr_arguments` for example:

```
lib.execute_using_packagelist(
    ...,
   .callr_arguments  = list(stdout = paste0('./execution_', gsub('\s|-|:', '_', format(Sys.time())), '
)
```

See the example below for a complete example. When you do this, it somehow swallows the first character of every `stderr` that is directly returned (also from `message` calls) when `run_quietly = FALSE`, but the log file seems intact.

## Value

Will return the outcome of your `func_handle`.

## Example

If you would like to log the outcomes, provide the .callr_arguments:

```
lib.execute_using_packagelist(
    packages_to_load   = c(package.a =  '0.1.0'),
    func_handle        = function() {an_important_value(); package_a1(5, 10)},
    .wait_for_response = TRUE,
   .callr_arguments  = list(stdout = paste0('./execution_', gsub('\s|-|:', '_', format(Sys.time())), '
    .run_quietly       = TRUE
)
```

Another more simple example:

```
lib.execute_using_packagelist(
    packages_to_load   = c(dplyr =  ''),
    func_handle        = function() {mtcars}
)
```

---

`lib.git_show_untracked` *List all un-tracked library folders*

---

### Description

List all un-tracked directories (libraries) within the multiversion library. The returned un-tracked directories are cleaned up and printed so that only the unique combinations of each library and it's version is shown once.

### Usage

```
lib.git_show_untracked(lib_location = lib.location())
```

### Arguments

| | |
|---|---|
| `lib_location` | By default the default library path obtained with `lib.location()`. |

### Value

No return value, is called for the printed feedback. Will show which packages inside the library are not yet tracked by git (when that is desired). It is recommended to track packages with git so that

---

`lib.install` *Install packages and tarballs into R_MV_library*

---

### Description

This family of functions can help with installing packages without the risk of installing every minor package improvement as soon as it is released.

1. `lib.install_tarball` can install a tarball based on the tarball location and it's dependencies (like `c(dplyr = '> 5.0')`).
2. `lib.install_if_not_compatible` can install CRAN package depending on a condition. This is especially useful (and used on the background) for installing the dependencies for the tarball installation.
3. `lib.install` can install CRAN packages into the R_MV_library, which in return is used by `lib.install_if_not_compatible`.

## Usage

```
lib.install(
  package_names = NULL,
  lib_location = lib.location(),
  install_temporarily = FALSE,
  allow_overwrite_on_convert = FALSE,
  quiet = TRUE,
  cran_url = "http://cran.us.r-project.org"
)

lib.install_if_not_compatible(
  package_conditions,
  lib_location = lib.location(),
  install_temporarily = FALSE,
  allow_overwrite_on_convert = FALSE,
  quiet = TRUE,
  cran_url = "http://cran.us.r-project.org"
)

lib.install_tarball(
  tarball,
  dependencies = c(),
  lib_location = lib.location(),
  install_temporarily = FALSE,
  allow_overwrite_on_convert = c("tarball", "dependencies"),
  cran_url = "http://cran.us.r-project.org"
)
```

## Arguments

| | |
|---|---|
| `package_names` | Provide a vector of package names. A version cannot be supplied. |
| `lib_location` | The folder where this package can be installed. The package will first be installed in a temporary install folder `<multiversion lib>/TEMP_install_location` indicated by the `lib.location_install_dir()` function. If `install_temporarily` is set to FALSE (the default), the installed package(s) is moved to the lib_location automatically. |
| `install_temporarily` | If FALSE, the installed packages are moved to the R_MV_library, specified by the `lib_location` argument, automatically. Otherwise it is necessary to run `lib.convert()` manually after the installation into the temporary folder finished. When multiple tarballs are provided, this is set to `FALSE` with no warning. |
| `allow_overwrite_on_convert` | Can be used if you are experimenting and you would like to overwrite the installed (tarball) package. Only makes sense with `install_temporarily` on `FALSE`. See details below. |
| `quiet` | Will affect `install.packages(...,quiet = quiet)`. |
| `cran_url` | Will be passed trough to the install.packages command. |

| | |
|---|---|
| `package_conditions` | Provide a vector of package name/'version condition' specifications. See section 'limitations for `package_conditions`'. |
| `tarball` | The complete path to the tarball file that you would like to install. |
| `dependencies` | Provide the dependencies like a package version combination: `c(dplyr = '>= 0.5',data.table = '',R6 = '0.1.1')`. Note that all dependencies must refer to packages on CRAN. Otherwise install the dependency manually somewhere and use `lib.convert` to include it. |

## Value

Nothing is returned, this function is called for it's side-effect of installing a package in the multi-version library.

## limitations for `package_conditions`

All version specifications are allowed except for the exact version indication (e.g. don't provide `c(dplyr = '1.2.3')`). It is allowed to provide no specification, which will match any installed version of that package. **If the condition is met, the package is skipped**, which is the desired behavior for dependencies. For an empty condition (e.g. `c(dplyr = '')`), it will only install the package when no version is installed at all.

## Allow overwrite on convert

When an installed package is converted to the R_MV_library, it would normally show that it failed to copy the packages of which that version was already present. This means that these packages were already converted from the temporary library to the R_MV_library structure before, and no `lib.clean_install_dir()` was performed yet. In case you are experimenting with a self made tarball package, and you are developing the package within the same package version, it is some times desired to overwrite the already present installed package with a new installation. For CRAN packages, this options doesn't make sense.

Only for `lib.install_tarball`, the options `TRUE`, `FALSE`, and additionally `"tarball"` `"dependencies"` are allowed. 'dependencies' will affect all packages that are in the temporary installation location except for the tarball package. 'tarball' will only overwrite the tarball package.

## Installing temporarily

Installing a package temporarily gives you the opportunity to test the package before adding it to the multiversion library structure. Loading packages, including those in the temporary library (`lib.location_install_dir()`) can be done using: `lib.load(...,also_load_from_temp_lib = TRUE)`.

## Note

To clean up the installation directory, run `lib.clean_install_dir()`.

## lib.installed_packages

*Show the complete library content.*

### Description

Use to print all available packages in the R_MV_library with all their versions including their dependencies. Simply performs a call to `lib.dependsOnMe(all)`.

### Usage

```
lib.installed_packages(lib_location = lib.location(), dont_print = FALSE)
```

### Arguments

| | |
|---|---|
| `lib_location` | The R_MV_library location. |
| `dont_print` | When true, will not print anything, but will expect you to make use of the invisibly returned package character vector. |

### Value

It returns a special character array with package:version names for every package and package version in the library.

## lib.is_basepackage

*Check if a package belongs to the standard R (base) packages.*

### Description

To check if the package is a base package, we look it up among all packages in the `.Library` directory (`list.dirs(.Library,full.names = FALSE,recursive = FALSE)`). We cannot version control packages which are located in this library since the `.Library` will always be added to the `.libPaths`. For base packages, this is acceptable, but it appears that this directory is not always as clean as we would wish. Because of this reason, we do not check the more widely accepted `rownames(installed.packages(priority="base"))`.

### Usage

```
lib.is_basepackage(packageName)
```

### Arguments

| | |
|---|---|
| `packageName` | The package name to check. |

### Value

Returns logical indicating if the provided package name is a base package or not.

## `lib.load` *Load package from R_MV_library*

### Description

There are two ways you can provide a package or vector of packages that need to be loaded:
1: just provide them directly (the `...` input). All not recognized named variables will be interpreted as package names or (if it's a named argument) as a package name=version combination. `lib.load(DBI = '0.5',assertthat,R6)` 2: provide the `loadPackages` input in the following way: `lib.load(loadPackages = c(DBI = '0.5',assertthat = '',R6 = ''))`

If an empty string e.g. `dplyr = ''`, or only the package name is specified, one of two things will happen: - if one version is available, this one is used. - if multiple versions are available, the first or last version is loaded depending on the 'pick.last' value (FALSE by default).

if >= or > is used, as in `dplyr = '>= 2.5'`, it will decide for the first or last compatible version, depending on the 'pick.last' parameter. If another version is desired, please define it in the input list of packages to load, prior to the package that depends on it.

### Usage

```
lib.load(
  ...,
  loadPackages = NULL,
  lib_location = lib.location(),
  dry_run = FALSE,
  quietly = FALSE,
  verbose = FALSE,
  appendLibPaths = FALSE,
  pick.last = FALSE,
  also_load_from_temp_lib = FALSE,
  .packNameVersionList = c(),
  .skipDependencies = c()
)
```

### Arguments

| | |
|---|---|
| `...` | All packages and their versions you would like to load e.g. `lib.load(DBI = '0.5',assertthat = '',R6 = '',quietly = TRUE)`. Input names like `quietly` will be recognized and interpreted as expected. |
| `loadPackages` | Supports providing a named character vector of packages and their versions in the shape that is supported by all other functions in this package. e.g. `c(DBI = '0.5',assertthat = '',R6 = '')` |
| `lib_location` | The folder containing a structure where this package must load packages from. By default, it checks the environment variable `R_MV_LIBRARY_LOCATION` for this directory. |

| | |
|---|---|
| `dry_run` | Will make it perform a dry run. It will check all dependencies and if `appendLibPaths` it will add their paths to `.libPaths` but it will not load those packages. If the paths are added this way, you should be able to just call the located packages with `library(...)` |
| `quietly` | Indicates if the loading must happen silently. No messages and warnings will be shown if the value is set to true. |
| `verbose` | Indicates if additional information must be shown that might help with debugging the decision flow of this function. More specifically, when false, it will wrap 'library' calls in `suppressWarnings(suppressMessages(...))` and suppress unloading attempts. |
| `appendLibPaths` | When true, the path to every package that is loaded will be appended to `.libPath(...)`. That configured path is the location where `library()` will look for packages. For a usecase for this feature, see the description above. |
| `pick.last` | Changes the way a decision is made. In the scenario where a dependency of > or >= is defined, multiple versions may be available to choose from. By default, the lowest compliant version is chosen. Setting this to true will choose the highest version. |
| `also_load_from_temp_lib` | when true, it will also load packages from the temporary installation directory (created when packages are installed in the R_MV_library). Can be usefull when installing using: `lib.install("new package!",install_temporarily = TRUE)`. |
| `.packNameVersionList` | See main description. Should be left blank. |
| `.skipDependencies` | See main description. Should be left blank. |

## Details

Dependencies are checked by recursively running this function with `dry_run = TRUE`. Then the paths of the found dependencies are temporarily appended (`.libPaths()`) when the actual package is loaded. This makes that dependencies are not loaded automatically, but are added to the namespace. To access a dependency directly, load it explicitly. Because the `.libPaths()` does not include the package it's location, this still needs to be done by `lib.load`.
In other words, dependencies are remembered, but not loaded.

Using `dry_run` will show the packages that will be used and will crash when no option is feasible (not installed or not compliant packages). If you are trying to setup a proper `lib.load` call, it is always a good idea to work with dry_run's. Once an incorrect package has been loaded, it is very likely you will have to restart your R session to unload it (`Ctrl + shift + F10`). Unloading packages in R often leaves traces.

The .libPaths of specific package versions can be appended when using 'appendLibPaths = TRUE'. Afterwards, the normal `library` call can be used to load the package since it's path is in the `.LibPaths` vector. For example:
`lib.load(c(dplyr = '0.5.0'),dry_run = TRUE,appendLibPaths = TRUE) library(dplyr)`

How this works is that `dry_run` skips the loading step, and `appendLibPaths` adds the paths of dplyr and it's dependencies to `.libPaths`, which make a `library` call work.

One reason to use `appendLibPaths = TRUE` is to make these packages accessible by a new 'child' R session. This is the case if `devtools::test()` is ran by using `cntrl + shift + TRUE` in Rstudio. When running it directly, it will use the packages it can find in the available libraries (`.libPath()`) and return an error if they cannot be found.

The inputs .packNameVersionList [vector of named versions] and .skipDependencies [vector of names] can be left blank in general. They are used internally and might be deprecated in the future.

## Value

Will return a named character vector indicating which version of which package is loaded (or will be loaded, when dry_run = TRUE). In general, this function is called for it's side effect. It will load specific versions of specific packages from a special `multiversion` library.

## Major version differences

By default, when chosing the right version to load, only versions are looked up within the same major version. For example, when `pick.last = TRUE`, the version `'> 15.3.0'` is requested and the versions `c('15.5.0','15.9.0','16.0.0')` are available, the version `15.9.0` is chosen. When a requested (dependency) version `'>= 0.5'` is provided, and only the versions `c('0.4.0','1.5.0','1.7.0')` are available, it will throw a warning that the first available version is a major release higher, and pick `'1.5.0'` or `'1.7.0'` depending on the `pick.last` value.

This behavior can be disabled by setting `options(mv_prefer_within_major_version = 'no')`.

## Base packages

The packages within the directory returned by `.Library` are considered 'base packages'. Of these, only one version can exist, and these cannot be included in the multiversion library.

## Problem solving

If you receive the error "`cannot unload ...`" it means that it tries to load a package, but another version is already loaded. To unload this other (older) version, run detach(package = '...'). If it is a dependency of an other package, you will receive this error. Try restarting your RStudio with a clean workspace (environment). If that doesn't help, the only workaround (when using this in R studio) is to close your Rstudio session (NOTE: save your unsaved process before proceeding!!), rename (or remove) the folder "`YourRProject/.Rproj.user/.../sources/prop`" and start Rstudio again. If it doesn't work, try "`/sources/per`" also. Where the `...` stands for a hash that is used in the current session e.g. `/F3B1663E/`. After this, the packages should be unloaded and you should be able to load a new batch of packages. Most times it will suffice to clear the workspace (environment) and reload the project while saving the empty environment.

## `lib.load_namespaces` *Load namespaces*

### Description

Load (but do not attach) the namespaces of a list of packages.

### Usage

```
lib.load_namespaces(
  packages_to_load_in_ns,
  lib_location = lib.location(),
  additional_lib
)
```

### Arguments

| | |
|---|---|
| `packages_to_load_in_ns` | A named character vector with package names and their version indication (e.g. ‘c(dplyr = ’>= 0.4.0’, ggplot = ”)‘). |
| `lib_location` | The folder which contains the multiversion library. By default, it checks the environment variable `R_MV_LIBRARY_LOCATION` to find this directory, see `lib.location()`. |
| `additional_lib` | A single or multiple paths that must be used in addition to the lib_location for looking up the packages. Non existing paths are silently ignored. |

## `lib.location` *The R_MV_library location.*

### Description

This function will look for the environment variable `R_MV_LIBRARY_LOCATION` indicating the R_MV_library location. Alternatively you can provide a path for this session only, using `lib.location(yourPath)`. This will set the environment variable for this session. (You might want to consider to add this to your `.Rprofile` file, see `?Startup`)

### Usage

```
lib.location(set_session_path)
```

### Arguments

| | |
|---|---|
| `set_session_path` | (optional) If no environment variable has been set to indicate the library location, You can call this function and let it set the environment variable for this session only. |

### Value

When a path is provided, this will be stored as the multiversion library location to use during this session (via the environment variable "R_MV_LIBRARY_LOCATION"). In all cases, it will return the location of the multiversion library. When it cannot be found, it will return an error indicating what to do.

---

## `lib.location_install_dir` *Temporary directory location.*

---

### Description

Indicates the default directory for initially installing a package before it is 'converted' to the final multiversion library structure (see: `lib.convert()`). This folder can be cleaned up using `cleanTempInstallFolder()` after installing the package succeeded. This is not done automatically but won't influence the installation of other packages.

### Usage

```
lib.location_install_dir(lib_location = lib.location(), do_create = TRUE)
```

### Arguments

| | |
|---|---|
| `lib_location` | By default the default library path obtained with `lib.location()`. |
| `do_create` | When it doesn't exist yet, create the folder. |

### Value

The temporary folder location where packages are installed before they are moved to their final location in the multiversion library. When `do_create == TRUE`, the folder will be created when it does not yet exist.

---

## `lib.package_version_loaded` *Check the versions of an already loaded package.*

---

### Description

This works for both packages within the R_MV_library (which are loaded) and for packages outside the library.

### Usage

```
lib.package_version_loaded(packageNames, exclude_not_loaded = TRUE)
```

## Arguments

| | |
|---|---|
| `packageNames` | The name or a vector of names of the packages for which to obtain the version. |
| `exclude_not_loaded` | If true, the default, it will not try to find a 'loaded version' of a package that is not loaded. |

## Value

A named character vector with the package name(s) and it's version that is loaded.

---

`lib.packs_str2vec` *Parse single string to named character vector.*

---

## Description

Parses a string shaped like:
`assertthat (>= 0.1),R6 (>= 2.1.2),Rcpp (>= 0.12.3),tibble (>= 1.2),magrittr (>= 1.5),lazyeval (>= 0.2.0),DBI (>= 0.4.1)`
to match the normal package name/version layout like:
`assertthat R6 Rcpp tibble magrittr lazyeval DBI`
`">= 0.1" ">= 2.1.2" ">= 0.12.3" ">= 1.2" ">= 1.5" ">= 0.2.0" ">= 0.4.1"`

Used by `lib.dependencies_online` to interpret the by CRAN provided dependencies, used to parse the 'vc_override_dependencies.txt' files and the dependencies mentioned in the 'DESCRIPTION' files of the installed packages.

## Usage

```
lib.packs_str2vec(deps)
```

## Arguments

| | |
|---|---|
| `deps` | A string of length one with the format shown in the description. This will be converted to a named character vector. |

## Value

Returns a character vector with the packages and their versions that it could derive from the single provided string.

## `lib.packs_vec2str` *Convert package name/version vector to single string.*

### Description

Used to print a set of package names and their version criteria in a way that `lib.packs_str2vec()` can parse it again to a package vector. This way we can list the dependencies of a function easily and support command line interaction for example.

### Usage

```
lib.packs_vec2str(x, do_return = FALSE)
```

### Arguments

| | |
|---|---|
| `x` | A named character vector with package names/versions. `c(dplyr = '>= 1.5.0',data.table = '')` |
| `do_return` | If FALSE (the default) the package sting is printed, if TRUE, it is returned as a character string and not printed. |

### Value

When do_return = TRUE, returns a character string that describes a vector with packages and their version specifications like `"dplyr (>= 1.5.0),data.table"`. When FALSE, it prints this string and returns nothing.

## `lib.printVerboseLibCall` *Print example* `lib.load` *call.*

### Description

Prints the library call that you can use, based on a name/version input vector.

### Usage

```
lib.printVerboseLibCall(packNameVersion, .forceToPrint = FALSE)
```

### Arguments

| | |
|---|---|
| `packNameVersion` | A named character vector with package names and their version indication (e.g. ‘c(dplyr = ’>= 0.4.0’, ggplot = ”)‘). |
| `.forceToPrint` | For testing, I need to be able to overrule the ’interactive()’ criteria for printing this example library call. |

## lib.set_libPaths *Set* `.libPaths()` *to the provided version specific package locations.*

### Description

Adds `.Library` and the paths of the specific versions of the provided packages that are specified (and likely loaded before) to the `.libPaths`. Note that this function will erase any current `.libPaths()` configuration silently.

### Usage

```
lib.set_libPaths(packNameVersion, lib_location, additional_lib_paths = c())
```

### Arguments

| | |
|---|---|
| `packNameVersion` | A named character vector with package names and their version indication (e.g. `c(dplyr = '>= 0.05',ggplot = '')`). Or the special string 'all', which will add the paths of all directories of the latest versions of every package in the R_MV_library. The path that is appended to the `.libPaths()` is constructed based on the name and version provided. |
| `lib_location` | The multiversion library location path (no default configured here!). |
| `additional_lib_paths` | Any additional `.libPaths()` that needs to be set. Namely used for the temporary installation directory. |

### Value

The old `.libPaths()` content is returned invisibly.

## normPath *Normalize path with backslashes.*

### Description

This short-hand function normalizes the path and makes sure only forward slashes are used. Other slashes are not usable in `grepl` statements directly for example, the '\' is parsed to '\' before being used as regexp.

### Usage

```
normPath(path)
```

### Arguments

| | |
|---|---|
| `path` | The path which needs to be normalized. Will make `C:/PROGRA~1/R/R-33~1.1/library` into `C:/Program Files/R/R-3.3.1/library`. |

## `raw_input_parser` *Parse direct unquoted input to package name/version vector.*

### Description

Converts input like `lib.load(hoi = 3.4,hai = '>= 7.9.2',FIETS)`
to a named character vector like `c(hoi = '3.4',hai = '>= 7.9.2',FIETS = '')`
which is compatible with all code that follows.

Must be called like `raw_input_parser(as.list(match.call()),c('named_param1','named_param2','named_param3`
It will return all (name) value pairs if values are available excluding the named parameters provided in the second argument.

### Usage

```
raw_input_parser(arguments, varnames_to_exclude)
```

### Arguments

| | |
|---|---|
| `arguments` | The `as.list(match.call())` list returned from the calling function. It creates a list of all provided arguments. |
| `varnames_to_exclude` | A character vector with var names to exclude. Normally that includes all arguments after `...`. |

## `strRemain` *Removes pattern A and pattern B from a string.*

### Description

Removes pattern A and pattern B from a string.

### Usage

```
strRemain(patA, patB, str)
```

### Arguments

| | |
|---|---|
| `patA` | The first regex pattern to remove from the string. |
| `patB` | The second regex pattern to remove from the remaining of the first removal. |
| `str` | The string that needs cleaning up. |

---

`unique_highest_package_versions`

*Create unique list of highest package versions.*

---

## Description

Creates a vector with the unique set of with package name = versions and will keep the highest version when multiple versions of one package are defined.

## Usage

```
unique_highest_package_versions(packNameVersion, return_as_df = FALSE)
```

## Arguments

| | |
|---|---|
| `packNameVersion` | provide a package name list like so: `c(dplyr = '0.5.0',R6 = '',R6 = 0.5)` |
| `return_as_df` | FALSE if the output should remain a structured dataframe, or if it should return a named character vector.<br>To get a feel with the function, you can try:<br>`multiversion:::unique_highest_package_versions(`<br>`  c(pack.a = '0.1.0', pack.c = '5.2', package.b = '1.9',  pack.c = '99.99'))` |

---

`with_safe_package_tester`

*Create a safe environment in which certain expressions can be tested*

---

## Description

Will reset the .libPaths, the 'R_MV_LIBRARY_LOCATION' environment variable to their old values and will unload 'package.a' till 'package.f' when finishing the execution.

## Usage

```
with_safe_package_tester(expr, also_clean_install_dir = FALSE)
```

## Arguments

| | |
|---|---|
| `expr` | The expression that needs to be evaluated in this protected environment. |
| `also_clean_install_dir` | If `lib.clean_install_dir()` must be run before and after the test. |

**Details**

Before execution it will set the following values:

1. .libPaths - will be set to .Library only.
2. `R_MV_LIBRARY_LOCATION` - will contain '../test_library/' or 'tests/test_library/' depending on the current directory.

# Index