Package 'box'

October 12, 2022

Title Write Reusable, Composable and Modular R Code

Version 1.1.2

URL <https://klmr.me/box/>, <https://github.com/klmr/box>

BugReports <https://github.com/klmr/box/issues>

Description A modern module system for R. Organise code into hierarchical,
composable, reusable modules, and use it effortlessly across projects via a
flexible, declarative dependency loading syntax.

Depends R (>= 3.5.0)

Imports tools

License MIT + file LICENSE

Encoding UTF-8

Suggests devtools, knitr, rmarkdown, R6, rlang, roxygen2 (>= 7.0.2),
shiny, testthat (>= 3.0.3)

Enhances rstudioapi

VignetteBuilder knitr

RoxygenNote 7.1.2

NeedsCompilation yes

Author Konrad Rudolph [aut, cre] (<<https://orcid.org/0000-0002-9866-7051>>),
Michael Schubert [ctb] (<<https://orcid.org/0000-0002-6862-5221>>)

Maintainer Konrad Rudolph <konrad.rudolph@gmail.com>

Repository CRAN

Date/Publication 2022-05-11 17:00:02 UTC

R topics documented:

---

box   *An alternative module system for R*

Description

Use box::use(prefix/mod) to import a module, or box::use(pkg) to import a package. Fully qualified names are supported for nested modules, reminiscent of module systems in many other modern languages.

Using modules & packages

- [box::use](#)

Writing modules

Infrastructure and utility functions that are mainly used inside modules.

- [box::file](#)
- [box::name](#)
- [box::register_S3_method](#)
- [mod-hooks](#)

Interactive use

Functions for use in interactive sessions and for testing.

- [box::help](#)
- [box::unload](#), [box::reload](#)
- [box::set_script_path](#)
- [box::script_path](#)

Author(s)

Maintainer: Konrad Rudolph <konrad.rudolph@gmail.com> ([ORCID](#))

Other contributors:

- Michael Schubert <mschu.dev@gmail.com> ([ORCID](#)) [contributor]

See Also

Useful links:

- <https://klmr.me/box/>
- <https://github.com/klmr/box>
- Report bugs at <https://github.com/klmr/box/issues>

---

export   *Explicitly declare module exports*

Description

box::export explicitly marks a source file as a **box** module. If can be used as an alternative to the @export tag comment to declare a module's exports.

Usage

```
box::export(...)
```

Arguments

...	zero or more unquoted names that should be exported from the module.

Details

box::export can be called inside a module to specify the module's exports. If a module contains a call to box::export, this call overrides any declarations made via the @export tag comment. When a module contains multiple calls to box::export, the union of all thus defined names is exported.

A module can also contain an argument-less call to box::export. This ensures that the module does not export any names. Otherwise, a module that defines names but does not mark them as exported would be treated as a *legacy module*, and all default-visible names would be exported from it. Default-visible names are names not starting with a dot (.). Another use of box::export() is to enable a module without exports to use module event hooks.

Value

box::export has no return value. It is called for its side effect.

Note

The preferred way of declaring exports is via the @export tag comment. The main purpose of box::export is to explicitly prevent exports, by being called without arguments.

See Also

box::use for information on declaring exports via @export.

---

file   *Find the full file names of files in modules*

Description

Find the full file names of files in modules

Usage

```
box::file(..., module = current_module())
```

Arguments

`...`	character vectors of files or subdirectories inside a module; if none is given, return the root directory of the module
`module`	a module environment (default: current module)

Value

A character vector containing the absolute paths to the files specified in `...`.

Note

If called from outside a module, the current working directory is used.

This function is similar to `system.file` for packages. Its semantics differ in the presence of non-existent files: `box::file` always returns the requested paths, even for non-existent files; whereas `system.file` returns empty strings for non-existent files, or fails (if requested via the argument `mustWork = TRUE`).

See Also

[system.file](#)

---

help   *Display module documentation*

Description

`box::help` displays help on a module's objects and functions in much the same way [help](#) does for package contents.

Usage

```
box::help(topic, help_type = getOption("help_type", "text"))
```

Arguments

`topic`	either the fully-qualified name of the object or function to get help for, in the format `module$function`; or a name that was exported and attached from an imported module or package.
`help_type`	character string specifying the output format; currently, only `'text'` is supported.

Details

See the vignette at `vignette('box', 'box')` for more information about displaying help for modules.

Value

`box::help` is called for its side effect when called directly from the command prompt.

---

mod-hooks	*Hooks for module events*

Description

Modules can declare functions to be called when a module is first loaded.

Usage

```
.on_load(ns)

.on_unload(ns)
```

Arguments

`ns`	the module namespace environment

Details

To create module hooks, modules should define a function with the specified name and signature. Module hooks should *not* be exported.

When `.on_load` is called, the unlocked module namespace environment is passed to it via its parameter ns. This means that code in `.on_load` is permitted to modify the namespace by adding names to, replacing names in, or removing names from the namespace.

`.on_unload` is called when modules are unloaded. The (locked) module namespace is passed as an argument. It is primarily useful to clean up resources used by the module. Note that, as for packages, `.on_unload` is *not* necessarily called when R is shut down.

Legacy modules cannot use hooks. To use hooks, the module needs to contain an export specification (if the module should not export any names, specify an explicit, empty export list via `box::export()`.

Value

Any return values of the hook functions are ignored.

Note

The API for hook functions is still subject to change. In particular, there might in the future be a way to subscribe to module events of other modules and packages, equivalently to R package userhooks.

---

name   *Get a module's name*

Description

Get a module's name

Usage

```
box::name()
```

Value

box::name returns a character string containing the name of the module, or NULL if called from outside a module.

Note

Because this function returns NULL if not invoked inside a module, the function can be used to check whether a code is being imported as a module or called directly.

---

register_S3_method   *Register S3 methods*

Description

box::register_S3_method makes an S3 method for a given generic and class known inside a module.

Usage

```
box::register_S3_method(name, class, method)
```

Arguments

name	the name of the generic as a character string.
class	the class name.
method	the method to register.

Details

Methods for generics defined in the same module do not need to be registered explicitly, and indeed *should not* be registered. However, if the user wants to add a method for a known generic (defined outside the module, e.g. print), then this needs to be made known explicitly.

See the vignette at vignette('box', 'box') for more information about defining S3 methods inside modules.

Value

register_S3_method is called for its side effect.

Note

Do not call registerS3method inside a module. Only use box::register_S3_method. This is important for the module's own book-keeping.

---

script_path | *Set the base path of the script*

Description

box::set_script_path(path) explicitly tells **box** the path of a given script from which it is called; box::script_path() returns the previously set path.

Usage

```
box::set_script_path(path)

box::script_path()
```

Arguments

path   character string containing the relative or absolute path to the currently executing R code file, or NULL to reset the path.

Details

box needs to know the base path of the topmost calling R context (i.e. the script) to find relative import locations. In most cases, **box** can figure the path out automatically. However, in some cases third-party packages load code in a way in which **box** cannot find the correct path of the script any more. box::set_script_path can be used in these cases to set the path of the currently executing R script manually.

Value

Both box::script_path and box::set_script_path return the previously set script path, or NULL if none was explicitly set. box::set_script_path returns its value invisibly.

Note

> **box** *should* be able to figure out the script path automatically. Using box::set_script_path should therefore never be necessary. [Please file an issue](#) if you encounter a situation that necessitates using box::set_script_path!

Examples

```
box::set_script_path('scripts/my_script.r')
```

---

unload	*Unload or reload a given module*

Description

> Given a module which has been previously loaded and is assigned to an alias mod, unload(mod unloads it; reload(mod) unloads and reloads it from its source.

Usage

```
box::unload(mod)

box::reload(mod)
```

Arguments

> mod   a module object to be unloaded or reloaded

Details

> Unloading a module causes it to be purged from the internal cache such that the next subsequent box::use declaration will reload the module from its source. box::reload unloads and reloads the specified modules and all its transitive module dependencies. box::reload is *not* merely a shortcut for calling box::unload followed by box::use, because box::unload only unloads the specified module itself, not any dependent modules.

Value

> box::unload and box::reload are called for their side effect. They do not return anything.

Note

> Any other references to the loaded modules remain unchanged, and will (usually) still work. Unloading and reloading modules is primarily useful for testing during development, and *should not be used in production code:* in particular, unloading may break other module references if the .on_unload hook unloaded any binary shared libraries which are still referenced.

> box::unload and box::reload come with a few restrictions. box::unload attempts to detach names attached by the corresponding box::use call. box::reload attempts to re-attach these same names. This only works if the corresponding box::use declaration is located in the same scope.

box::unload will execute the .on_unload hook of the module, if it exists. box::reload will re-execute the .on_load hook of the module and of all dependent modules during loading (after executing the corresponding .on_unload hooks during unloading).

See Also

box::use, module hooks

---

use	*Import a module or package*

Description

box::use imports one or more modules and/or packages, and makes them available in the calling environment.

Usage

```
box::use(prefix/mod, ...)

box::use(pkg, ...)

box::use(alias = prefix/mod, ...)

box::use(alias = pkg, ...)

box::use(prefix/mod[attach_list], ...)

box::use(pkg[attach_list], ...)
```

Arguments

...	further import declarations
prefix/mod	a qualified module name
pkg	a package name
alias	an alias name
attach_list	a list of names to attached, optionally witha aliases of the form alias = name; or the special placeholder name ...

Details

box::use(...) specifies a list of one or more import declarations, given as individual arguments to box::use, separated by comma. Each import declaration takes one of the following forms:

prefix/mod: Import a module given the qualified module name *prefix/mod* and make it available locally using the name *mod*. The *prefix* itself can be a nested name to allow importing specific submodules. *Local imports* can be specified via the prefixes starting with . and .., to override the search path and use the local path instead. See the 'Search path' below for details.

pkg: Import a package *pkg* and make it available locally using its own package name.

alias = *prefix* / *mod* **or** *alias* = *pkg*: Import a module or package, and make it available locally using the name *alias* instead of its regular module or package name.

prefix / *mod*[*attach_list*] **or** *pkg*[*attach_list*]: Import a module or package and attach the exported symbols listed in *attach_list* locally. This declaration does *not* make the module/package itself available locally. To override this, provide an alias, that is, use *alias* = *prefix* / *mod*[*attach_list*] or *alias* = *pkg*[*attach_list*].

> The *attach_list* is a comma-separated list of names, optionally with aliases assigned via alias = name. The list can also contain the special symbol ..., which causes *all* exported names of the module/package to be imported.

See the vignette at vignette('box', 'box') for detailed examples of the different types of use declarations listed above.

Value

box::use has no return value. It is called for its side effect.

Import semantics

Modules and packages are loaded into dedicated namespace environments. Names from a module or package can be selectively attached to the current scope as shown above.

Unlike with [library](), attaching happens *locally*, i.e. in the caller's environment: if box::use is executed in the global environment, the effect is the same. Otherwise, the effect of importing and attaching a module or package is limited to the caller's local scope (its environment()). When used *inside a module* at module scope, the newly imported module is only available inside the module's scope, not outside it (nor in other modules which might be loaded).

Member access of (non-attached) exported names of modules and packages happens via the $ operator. This operator does not perform partial argument matching, in contrast with the behavior of the $ operator in base R, which matches partial names.

Export specification

Names defined in modules can be marked as *exported* by prefixing them with an @export tag comment; that is, the name needs to be immediately prefixed by a comment that reads, verbatim, #' @export. That line may optionally be part of a **roxygen2** documentation for that name.

Alternatively, exports may be specified via the [box::export]() function, but using declarative @export tags is generally preferred.

A module which has not declared any exports is treated as a *legacy module* and exports *all* default-visible names (that is, all names that do not start with a dot (.). This usage is present only for backwards compatibility with plain R scripts, and its usage is *not recommended* when writing new modules.

To define a module that exports no names, call box::export() without arguments. This prevents the module from being treated as a legacy module.

Search path

Modules are searched in the module search path, given by `getOption('box.path')`. This is a character vector of paths to search, from the highest to the lowest priority. The current directory is always considered last. That is, if a file 'a/b.r' exists both locally in the current directory and in a module search path, the local file './a/b.r' will *not* be loaded, unless the import is explicitly declared as `box::use(./a/b)`.

Modules in the module search path *must be organised in subfolders*, and must be imported fully qualified. Keep in mind that `box::use(name)` will *never* attempt to load a module; it always attempts to load a package. A common module organisation is by project, company or user name; for instance, fully qualified module names could mirror repository names on source code sharing websites (such as GitHub).

Given a declaration `box::use(a/b)` and a search path '*p*', if the file '*p*/a/b.r' does not exist, **box** alternatively looks for a nested file '*p*/a/b/__init__r' to load. Module path names are *case sensitive* (even on case insensitive file systems), but the file extension can be spelled as either '.r' or '.R' (if both exist, .r is given preference).

The module search path can be overridden by the environment variable R_BOX_PATH. If set, it may consist of one or more search paths, separated by the platform's path separator (i.e. ; on Windows, and : on most other platforms).

The *current directory* is context-dependent: inside a module, the directory corresponds to the module's directory. Inside an R code file invoked from the command line, it corresponds to the directory containing that file. If the code is running inside a **Shiny** application or a **knitr** document, the directory of the execution is used. Otherwise (e.g. in an interactive R session), the current working directory as given by `getwd()` is used.

Local import declarations (that is, module prefixes that start with ./ or ../) never use the search path to find the module. Instead, only the current module's directory (for ./) or the parent module's directory (for ../) is looked at. ../ can be nested: ../../ denotes the grandparent module, etc.

S3 support

Modules can contain S3 generics and methods. To override known generics (= those defined outside the module), methods inside a module need to be registered using `box::register_S3_method`. See the documentation there for details.

Module names

A module's full name consists of one or more R names separated by /. Since `box::use` declarations contain R expressions, the names need to be valid R names. Non-syntactic names need to be wrapped in backticks; see Quotes.

Furthermore, since module names usually correspond to file or folder names, they should consist only of valid path name characters to ensure portability.

Encoding

All module source code files are assumed to be UTF-8 encoded.

See Also

box::name and box::file give information about loaded modules. box::help displays help for a module's exported names. box::unload and box::reload aid during module development by performing dynamic unloading and reloading of modules in a running R session. box::export can be used inside a module to declare module exports.

Examples

```
local({
    # Set the module search path for the example module.
    old_opts = options(box.path = system.file(package = 'box'))
    on.exit(options(old_opts))

    # Basic usage
    # The file `box/hello_world.r` exports the functions `world` and `bye`.
    box::use(box/hello_world)
    hello_world$hello('Robert')
    hello_world$bye('Robert')

    # Using an alias
    box::use(world = box/hello_world)
    world$hello('John')

    # Attaching exported names
    box::use(box/hello_world[hello])
    hello('Jenny')
    # Exported but not attached, thus access fails:
    try(bye('Jenny'))

    # Attach everything, give 'world' an alias:
    box::use(box/hello_world[hi = hello, ...])
    hi('Eve')
    bye('Eve')
})
```

Index