Package ‘modules’

August 25, 2023

Title Self Contained Units of Source Code
Version 0.12.0
Description Provides modules as an organizational unit for source code. Modules enforce to be more rigorous when defining dependencies and have a local search path. They can be used as a sub unit within packages or in scripts.

BugReports https://github.com/wahani/modules/issues
URL https://github.com/wahani/modules
ByteCompile TRUE
Depends R (>= 3.2.0)
Imports utils
License MIT + file LICENSE
Encoding UTF-8
Suggests testthat, devtools, knitr, lintr, rmarkdown, parallel
RoxygenNote 7.1.2
Collate 'amodule.R' 'NAMESPACE.R' 'getSearchPath.R' 'class.R' 'depend.R' 'export.R' 'expose.R' 'extend.R' 'import.R' 'module-class.R' 'module-coercion.R' 'module-helper.R' 'module.R' 'use.R' 'testModule.R' 'base-override.R'

VignetteBuilder knitr
NeedsCompilation no
Author Sebastian Warnholz [aut, cre]
Maintainer Sebastian Warnholz <wahani@gmail.com>
Repository CRAN
Date/Publication 2023-08-25 13:50:05 UTC
R topics documented:

amodule .............................................. 2
as.module .......................................... 3
depend ............................................... 4
export ............................................... 4
expose ............................................... 6
extend ............................................... 7
getSearchPath ...................................... 8
import ............................................. 8
module .............................................. 10
use .................................................. 11

Index 14

amodule

Define Augmented and Parameterized Modules

**Description**

`amodule` is a wrapper around `module` and changes the default environment to which the module connects. In contrast to `module`, the top enclosing environment here is always `baseenv`. The second important difference is that the environment in which a module is created has meaning: all objects are made available to the module scope. This is what is meant by *augmented* or *parameterized*. Best practice for the use of this behavior is to return these modules from functions.

**Usage**

```r
amodule(expr = {
}, envir = parent.frame(), enclos = baseenv(), class = NULL)
```

**Arguments**

- `expr` *(expression)* a module declaration, same as `module`
- `envir` *(environment)* environment used to detect 'parameters'
- `enclos` *(environment)* the top enclosing environment of the module scope.
- `class` *(character)* the module can have a class attribute for consistency. If you rely on S3 dispatch, e.g. to override the default print method, you should set this value explicitly.

**Examples**

```r
Constructor <- function(dependency) {
  amodule({
    fun <- function(...) dependency(...)
    })
}
instance <- Constructor(identity)
```
as.module

instance$fun(1)

---

as.module  Coercion for Modules

Description

Interfaces to and from modules.

Usage

as.module(x, ...)

## S3 method for class 'character'
as.module(x, topEncl = baseenv(), reInit = TRUE, ..., envir = parent.frame())

## S3 method for class 'module'
as.module(x, reInit = TRUE, ...)

Arguments

x  something which can be coerced into a module. character are interpreted as file / folder names.

...  arguments passed to parse

topEncl  (environment) the root of the local search path. It is tried to find a good default via autoTopEncl.

reInit  (logical) if a module should be re-initialized

envir  (environment) the environment from where module is called. Used to determine the top level environment and should not be supplied by the use.

Examples

# as.module is used by 'use' so see the vignette for examples:
## Not run:
vignette("modulesInR", "modules")

## End(Not run)
depend  

*Declare dependencies of modules*

**Description**

This function will check for a dependency and tries to make it available if it is not. This is a generic function. Currently only a default method exists which assumes a package name as argument. If a package is not installed `depend` tries to install it.

**Usage**

```r
depend(on, ...)  
## Default S3 method:
depend(on, version = "any", libPath = NULL, ...)
```

**Arguments**

- `on`  
  (character) a package name
- `...`  
  arguments passed to `install.packages`
- `version`  
  (character) a version, defaults to 'any'
- `libPath`  
  (character | NULL) a path to the library (folder where packages are installed)

**Value**

TRUE if dependency is available or successfully installed. An error if dependency can not be installed and is not available.

**Examples**

```r
# Depend on certain R version
depend("base", "3.0.0")
# Depend on package version
depend("modules", "0.6.0")
```

export  

*Export mechanism for modules*

**Description**

You can declare exports very much like the export mechanism in R packages: you define which objects from the module you make available to a user. All other objects are kept private, local, to the module.
Usage

```r
export(..., where = parent.frame())
```

Arguments

- `...`: (character, or unquoted expression) names to export from module. A character
  of length 1 with a leading `^` is interpreted as regular expression. Arguments
  can be named and used for renaming exports.

- `where`: (environment) typically the calling environment. Should only be relevant for
  testing.

Details

A module can have several export declarations, e.g. directly in front of each function definition.
That means: exports stack up. When you supply a regular expression, however, only one export
pattern should be declared. A regular expression is denoted, as a convention, as character vector of
length one with a leading `^`.

Examples

```r
module({
  export("foo")
  foo <- function() "foo"
  bar <- function() "bar"
})

module({
  export("foo")
  foo <- function() "foo"
  export("bar")
  bar <- function() "bar"
})

module({
  export("foo", "bar")
  foo <- function() "foo"
  bar <- function() "bar"
})

module({
  export("^f.*$")
  foo <- function() "foo"
  bar <- function() "bar"
})

module({
  export(bar = foo)
  foo <- function() "foo"
})
```
expose

Exposé module contents

Description

Use expose to copy the exported member of a module to the calling environment. This is useful for a simple reexport of member functions and generally for object composition.

Usage

expose(module, ..., reInit = TRUE, where = parent.frame())

Arguments

module (character | module) a module as file or folder name or a list representing a module.

... (character, or unquoted expression) elements to be exposed. Defaults to all.

reInit (logical) whether to re-initialize module. This is only relevant if a module has state which can be changed. This argument is passed to as.module.

where (environment) typically the calling environment. Should only be relevant for testing.

Details

You call this function for its side effects. It is a variation of use where instead of returning a module as return value, the elements are copied to the calling environment.

Examples

m1 <- module({
  foo <- function() "foo"
})
m2 <- module({
  bar <- function() "bar"
})
# Now we create a module with 'foo' and 'bar' as member functions.
m3 <- module({
  expose(m1)
  expose(m2)
})
m3$foo()
m3$bar()
extend

Extend existing module definitions

Description

*extend* can be used to extend an existing module definition. This can be very useful to write unit tests when they need to have access to private member functions of the module. This function breaks encapsulation of modules and should be used with great care. As a mechanism for reuse consider 'composition' using *expose* and *use*.

Usage

```r
extend(module, with)
```

Arguments

+ **module** (character | module) a module as file or folder name or a list representing a module.
+ **with** (expression) an expression to add to the module definition.

Details

A module can be characterized by its source code, the top enclosing environment and the environment the module has been defined in. *extend* will keep the latter two intact and only change the source code. That means that the new module will have the same scope as the module to be extended. *import*, *use*, and *export* declarations can be added as needed.

This approach gives access to all implementation details of a module and breaks encapsulation. Possible use cases are: unit tests, and hacking the module system when necessary. For general reuse of modules, consider using *expose* and *use* which are safer to use.

Since *extend* will alter the source code, the state of the module is ignored and will not be present in the new module. A fresh instance of that new module is returned and can in turn be extended and/or treated like any other module.

Examples

```r
m1 <- module({
  foo <- function() "foo"
})
m2 <- extend(m1, {
  bar <- function() "bar"
})
m1$foo()
m2$foo()
m2$bar()
# For unit tests consider using:
extend(m1, {
  stopifnot(foo() == "foo")
})
```
getSearchPath  
*Get the search path of an environment*

**Description**

Returns a list with the environments or names of the environments on the search path. These functions are used for testing, use `search` instead.

**Usage**

```r
getSearchPath(where = parent.frame())
getSearchPathNames(where = parent.frame())
getSearchPathContent(where = parent.frame())
getSearchPathDuplicates(where = parent.frame())
```

**Arguments**

*where*  
(environment | module | function) the object for the search path should be investigated. If we supply a list with functions (e.g. a module), the environment of the first function in that list is used.

**Examples**

```r
getSearchPath()
getSearchPathNames()
getSearchPathContent()

m <- module(
  export("foo")
  import("stats", "median")
  foo <- function() "foo"
  bar <- function() "bar"
)

gSearchPathContent(m)
```

---

**import**  
*Import mechanism for modules*

**Description**

You can declare imports similar to what we would do in a R package: we list complete packages or single function names from a package. These listed imports are made available inside the module scope.
**import**

### Usage

```r
import(from, ..., attach = TRUE, where = parent.frame())
importDefaultPackages(except = NULL, where = parent.frame())
```

### Arguments

- `from` (character, or unquoted expression) a package name
- `...` (character, or unquoted expression) names to import from package.
- `attach` (logical) whether to attach the imports to the search path.
- `where` (environment) typically the calling environment. Should only be relevant for testing.
- `except` (character | NULL) a character vector excluding any packages from being imported.

### Details

`import` and `use` can replace `library` and `attach`. However they behave differently and are only designed to be used within modules. Both will work when called in the `.GlobalEnv` but here they should only be used for development and debugging of modules.

`import` adds a layer to a local search path. More precisely to the calling environment, which is the environment supplied by `where`. It will alter the state of the calling environment. This is very similar to how the `library` function and the `search` path are constructed in base R. Noticeable differences are that we can choose to only import particular functions instead of complete packages. Further we do not have to mutate the calling environment by setting `attach` to `FALSE`. Regardless of the `attach` argument, `import` will return an environment with the imports and can be bound to a name. `library` will also load packages in the 'Depends' field of a package, this is something `import` will not do.

Only one `import` declaration per package is allowed. A second call to `import` will remove the previous one from the search path. Then the new import layer is added. If several smaller `import` declarations are desirable, use `attach = FALSE` and bind the return value of `import` to a name.

### Value

An `environment` is returned invisibly comprising the imports.

### Examples

```r
m <- module({
    # Single object from package
    import("stats", "median")
    # Complete package
    import("stats")
    # Without side-effects
    stats <- import("stats", attach = FALSE)
    median <- function(x) stats$median(x)
})
```
**Define Modules in R**

**Description**

Use `module` to define self contained organisational units. Modules have their own search path. `import` can be used to import packages. `use` can be used to import other modules. Use `export` to define which objects to export. `expose` can be used to reuse function definitions from another module.

**Usage**

```r
module(expr = {
}, topEncl = autoTopEncl(envir), envir = parent.frame())

autoTopEncl(where)
```

**Arguments**

- `expr` an expression
- `topEncl` (environment) the root of the local search path. It is tried to find a good default via `autoTopEncl`.
- `envir, where` (environment) the environment from where `module` is called. Used to determine the top level environment and should not be supplied by the use.

**Details**

topEncl is the environment where the search of the module ends. autoTopEncl handles the different situations. In general it defaults to the base environment or the environment from which `module` has been called. If you are using use or expose referring to a module in a file, it will always be the base environment. When identical(topenv(parent.frame())), globalenv()) is false it (most likely) means that the module is part of a package. In that case the module defines a sub unit within a package but has access to the packages namespace. This is relevant when you use the function `module` explicitly. When you define a nested module the search path connects to the environment of the enclosing module.

The use of `library`, `attach`, and `source` are discouraged within modules. They change the global state of an R session, the `.GlobalEnv`, and may not have the intended effect within modules. `import` and `use` can replace calls to `library` and `attach`. Both will work when called in the `.GlobalEnv` but here they should only be used for development and debugging of modules. `source` often is used to load additional user code into a session. This is what `use` is designed to do within modules. `use` will except files and folders to be used.

`export` will never export a function with a leading "." in its name.

`expose` is similar to `use` but instead of attaching a module it will copy all elements into the calling environment. This means that `exposed` functions can be (re-)exported.

`extend` can be used to extend an existing module definition. This feature is meant to be used by the module author. This can be very useful to write unit tests when they need to have access to private
member functions of the module. It is not safe as a user of a module to use this feature: it breaks encapsulation. When you are looking for mechanisms for reuse expose and use should be favoured.

Examples

## Not run:
vignette("modulesInR", "modules")

## End(Not run)

m <- module({
  fun <- function(x) x
})

m$fun(1)

m <- module(
  import("stats", "median")
  export("fun")

  fun <- function(x) {
    ## This is an identity function
    ## x (ANY)
    x
  }
}

m$fun
m

use

Use a module as dependency

Description

Use and/or register a module as dependency. The behaviour of use is similar to import but instead of importing from packages, we import from a module. A module can be defined in a file, or be an object.

Usage

use(module, ..., attach = FALSE, reInit = TRUE, where = parent.frame())
Arguments

- **module**: (character, module) a file or folder name, or an object that can be interpreted as a module: any list-like object would do.
- **...**: (character, or unquoted expression) names to use from module.
- **attach**: (logical) whether to attach the module to the search path.
- **reInit**: (logical) we can use a module as is, or reinitialize it. The default is to reinitialize. This is only relevant should the module be state-full.
- **where**: (environment) typically the calling environment. Should only be relevant for testing.

Details

`import` and `use` can replace `library` and `attach`. However they behave differently and are only designed to be used within modules. Both will work when called in the `.GlobalEnv` but here they should only be used for development and debugging of modules.

`use` adds a layer to a local search path if `attach` is `TRUE`. More precisely to the calling environment, which is the environment supplied by `where`. Regardless of the `attach` argument, `use` will return the module invisibly.

`use` supplies a special mechanism to find the argument `module`: generally you can supply a file name or folder name as character. You can also reference objects/names which 'live' outside the module scope. If names are not found within the scope of the module, they are searched for in the environment in which the module has been defined. This happens during initialization of the module, when the `use` function is called.

Modules can live in files. `use` should be used to load them. A module definition in a file does not need to use the `module` constructor explicitly. Any R script can be used as the body of a module.

When a folder is referenced in `use` it is transformed into a list of modules. This is represented as a nested list mimicking the folder structure. Each file in that folder becomes a module.

Examples

```r
m1 <- module(
  foo <- function() "foo"
)
m2 <- module(
  use(m1, attach = TRUE)
  bar <- function() "bar"
  m1foo <- function() foo()
)
m2$m1foo()
m2$bar()

## Not run:
someFile <- tempfile(fileext = ".R")
writelines("foo <- function() 'foo'", someFile)
m3 <- use(someFile)
m3$foo()
otherFile <- tempfile(fileext = ".R")
```
writeLines("bar <- function() 'bar'", otherFile)
m4 <- use(otherFile)
m4$bar()
m5 <- use(tempdir())
m5

## End(Not run)
Index

.GlobalEnv, 10
amodule, 2
as.module, 3, 6
attach, 9, 10, 12
autoTopEncl, 3, 10
autoTopEncl(module), 10
depend, 4
environment, 9
export, 4, 7, 10
expose, 6, 7, 10, 11
extend, 7, 7, 10
getSearchPath, 8
getSearchPathContent(getSearchPath), 8
getSearchPathDuplicates
  (getSearchPath), 8
getSearchPathNames(getSearchPath), 8
import, 7, 8, 10–12
importDefaultPackages(import), 8
install.packages, 4
library, 9, 10, 12
module, 2, 10, 12
parse, 3
search, 8, 9
source, 10
use, 6, 7, 9–11, 11