Package ‘modules’

July 15, 2018

Title Self Contained Units of Source Code

Version 0.7.0

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 http://wahani.github.io/modules

ByteCompile TRUE

Depends R (>= 3.2.0)

Imports stringr, utils

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Suggests disposables, testthat, devtools, knitr, lintr, rmarkdown, parallel

RoxygenNote 6.0.1

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'

VignetteBuilder knitr

NeedsCompilation no

Author Sebastian Warnholz [aut, cre]

Maintainer Sebastian Warnholz <wahani@gmail.com>

Repository CRAN

Date/Publication 2018-07-15 21:40:03 UTC
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)
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

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

# Depend on certain R version
depend("base", "3.0.0")

# Depend on package version
depend("modules", "0.6.0")

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

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

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

extend(module, with)

import(from, ..., attach = TRUE, where = parent.frame())

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

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

**Arguments**

... (character, or unquoted expression) names to import from package or names to export from module. For exports a character of length 1 with a leading "^" is interpreted as regular expression.

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

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

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.

from (character, or unquoted expression) a package name

attach (logical) whether to attach the module to the search path

expr, with an expression

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

eoir (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.

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 should only be used for development and debugging of modules.

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 a 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 consumer or user of a module to use this feature as it breaks encapsulation. When you are looking for mechanisms for reuse, expose and use should be favoured.
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

getSearchPath(where = parent.frame())

getSearchPathNames(where = parent.frame())

Arguments

where (environment)
getSearchPath

Examples
   getSearchPath()
   getSearchPathNames()
Index

amodule, 2
as.module, 3, 5
attach, 5
autoTopEncl, 3, 5
autoTopEncl (export), 4
depend, 3
export, 4
expose (export), 4
extend (export), 4
getSearchPath, 6
getSearchPathNames (getSearchPath), 6
import (export), 4
install.packages, 4
library, 5
module, 2
module (export), 4
parse, 3
search, 6
use (export), 4