Package 'svMisc'

June 30, 2018

Type Package

Version 1.1.0

Date 2018-06-10

Title SciViews - Miscellaneous Functions

Description Miscellaneous functions for SciViews or general use: manage a temporary environment attached to the search path for temporary variables you do not want to save() or load(), test if Aqua, Mac, Win, ... Show progress bar, etc.

Maintainer Philippe Grosjean <phgrosjean@sciviews.org>

Depends R (>= 2.13.0)

Imports utils, methods, stats, tools

Suggests rJava, tcltk, covr, knitr, testthat

License GPL-2

URL https://github.com/SciViews/svMisc, http://www.sciviews.org/SciViews-R

BugReports https://github.com/SciViews/svMisc/issues

RoxygenNote 6.0.1

VignetteBuilder knitr

NeedsCompilation no

Author Philippe Grosjean [aut, cre],
Romain Francois [ctb],
Kamil Barton [ctb]

Repository CRAN

Date/Publication 2018-06-30 17:04:09 UTC

R topics documented:

`svMisc-package` *SciViews - Miscellaneous functions*

Description

Miscellaneous functions for SciViews or general use. The svMisc package collects together a series of functions that are shared with svXXX packages.

Important functions

- `temp_env()` for unsing a temporary environment attached to the search path,
- `temp_var()` create the name of temporary variables,
- `capture_all()` to capture R output, errors, warnings and messages,
- `parse_text()` to parse any R expression, including partial or incorrect ones (fails gracefully).

about *Get information and help about* R *objects*

Description

Help obtained with this function is wider than with `help()`. If a man page is not found, it suggests related topics. If an object is an S3 generic function, it also lists all its known methods. Also, one can track the help page of an object even if its name is changed, by using the `src` or `srcfile` attribute of the object's comment. By the way, if the object has a comment, it is also displayed. This can be used as a quick and dirty way to provide short hints to custom objects. Finally, it is possible to track down the source of an object into a file with the `srcfile` attribute of its comment. In this case, it is the source file that is displayed. So, you can also further document your custom objects easily in their source files!

Usage

```
about(topic, ...)

"?"(type, topic)
```

Arguments

`topic`	The name of an object, or the topic to search for, if this is not the name of a known object.
`...`	Further arguments passed to `help()`.
`type`	First argument to `?`. If it is a dot, like `.?topic`, the second argument is a topic passed to the `about()` function. Otherwise, it is the first argument to restrict help pages, like `class`, `methods`, or `method`. See examples for how to use it.

Value

A string with the location of all objects named `topic` are found is returned invisibly.

See Also

`help()`, `help.search()`, `apropos()`

Examples

```
about("nonexisting") # Not found on search path, but help pages
about("htgdsfgfdsgf") # Not found anywhere
#library(tidyverse)
#about("group_by") # Just one page
#about("filter") # Several items
about("stats::filter") # OK
#about("dplyr::filter") # OK too
about("base::filter") # Not found there
# Objects with comment: print comment
```

```
vec <- structure(1:10, comment = "A simple vector")
about("vec")
# If there is a srcfile attribute in the comment, also display the file
# Hint: integrate some help in the header!
#library(data)
#(iris <- read(data_example("iris.csv")))
#about("iris")
# If the comment has a src attribute, change the topic to that one
#urchin <- read("urchin_bio", package = "data")
#about("urchin")
.?filter
.?stats::filter
```

`add_actions` *Add GUI elements like actions (menu items), icons, or methods in a predefined list*

Description

Manage lists of GUI actions, icons and methods.

Usage

```
add_actions(obj = get_actions(), text = NULL, code = NULL, state = NULL,
  options = NULL, replace = TRUE)

get_actions()

add_icons(obj = ".svIcons", icons, replace = TRUE)

add_methods(methods)

addActions(obj = get_actions(), text = NULL, code = NULL, state = NULL,
  options = NULL, replace = TRUE)

addIcons(obj = ".svIcons", icons, replace = TRUE)

addMethods(methods)
```

Arguments

`obj`	The name of the object in `SciViews:TempEnv` to manipulate.
`text`	The text of actions to add (label on first line, tip on other lines).
`code`	The R code of actions to add.
`state`	The default (intial) state of an action, as a succession of letters: `c` = checked, `u` = unchecked (default); `d` = disabled, `e` = enabled (default); `h` = hidden, `v` = visible (default). Default values are facultative. Ex: `udv` means: unchecked - disabled - visible and it equals to simply `d`, given the defaults for the other properties.

options	A character vector with other options to pass to the graphical toolkit for this action.
replace	Do we replace existing items in 'x'?
icons	The description of the icons to add.
methods	The list of methods to add (character string).

Value

The modified object is returned invisibly.

See Also

add_items(), obj_menu(), temp_env()

Examples

```
# This is useful to add actions, icons, descriptions, shortcuts or methods
# TODO: examples and use for functions add_actions(), add_icons() and
# add_methods()
```

---

batch *Run a function in batch mode*

Description

A function can be run in batch mode if it never fails (replace errors by warnings) and returns TRUE in case of success, or FALSE otherwise.

Usage

```
batch(items, fun, ..., show.progress = !is_aqua() && !is_jgr(),
  suppress.messages = show.progress, verbose = TRUE)
```

Arguments

items	The items (usually, arguments vector of character strings) on which to apply fun sequentially.
fun	The function to run (must return TRUE or FALSE and issue only warnings or messages to be a good candidate, batchable, function).
...	Further arguments to pass the fun.
show.progress	Do we show progression as item x on y... message? This uses the progress() function.
suppress.messages	Are messages from the batcheable function suppressed? Only warnings will be issued. Recommended if show.progress = TRUE.
verbose	Display start and end messages if TRUE (default).

Value

Returns invisibly the number of items that were correctly processed with attributes `items` and `ok` giving more details.

See Also

`progress()`

Examples

```
# Here is a fake batcheable process
fake_process <- function(file) {
  message("Processing ", file, "...")
  flush.console()
  Sys.sleep(0.5)
  if (runif(1) > 0.7) { # Fails
    warning("fake_process was unable to process ", file)
    invisible(FALSE)
  } else invisible(TRUE)
}

# Run it in batch mode on five items
files <- paste0("file", 1:5)
batch(files, fake_process)
```

`capture_all` *Run an R expression and capture output and messages in a similar way as it would be done at the command line*

Description

This function captures results of evaluating one or several R expressions the same way as it would be issued at the prompt in a R console. The result is returned in a character string. Errors, warnings and other conditions are treated as usual, including the delayed display of the warnings if `options(warn = 0)`.

Usage

```
capture_all(expr, split = TRUE, echo = TRUE, file = NULL,
  markStdErr = FALSE)

captureAll(expr, split = TRUE, echo = TRUE, file = NULL,
  markStdErr = FALSE)
```

Arguments

expr	A valid R expression to evaluate (names and calls are also accepted).
split	Do we split output, that is, do we also issue it at the R console too, or do we only capture it silently?
echo	Do we echo each expression in front of the results (like in the console)? In case the expression spans on more than 7 lines, only first and last three lines are echoed, separated by `[...]`.
file	A file, or a valid opened connection where output is sinked. It is closed at the end, and the function returns `NULL` in this case. If `file = NULL` (by default), a `textConnection()` captures the output and it is returned as a character string by the function.
markStdErr	If `TRUE`, stderr is separated from sddout by STX/ETX characters.

Value

Returns a string with the result of the evaluation done in the user workspace.

Note

If the expression is provided as a character string that should be evaluated, and you need a similar behaviour as at the prompt for incomplete lines of code (that is, to prompt for more), you should not parse the expression with `parse(text = "<some_code>")` because it returns an error instead of an indication of an incomplete code line. Use `parse_text("<some_code>")` instead, like in the examples bellow. Of course, you have to deal with incomplete line management in your GUI/CLI application... the function only returns `NA` instead of a character string.

See Also

`parse()`, `expression()`, `capture.output()`

Examples

```
writeLines(capture_all(expression(1 + 1), split = FALSE))
writeLines(capture_all(expression(1 + 1), split = TRUE))
writeLines(capture_all(parse_text("search()"), split = FALSE))
## Not run:
writeLines(capture_all(parse_text('1:2 + 1:3'), split = FALSE))
writeLines(capture_all(parse_text("badname"), split = FALSE))

## End(Not run)

# Management of incomplete lines
capt_res <- capture_all(parse_text("1 +")) # Clearly an incomplete command
if (is.na(capt_res)) cat("Incomplete line!\n") else writeLines(capt_res)
rm(capt_res)
```

compare_r_version *Compare current R version with a specified one*

Description

Determine if R is older (return -1), or not (return 0 if equal, or 1 if newer) than a given version number.

Usage

```
compare_r_version(version)

compareRVersion(version)
```

Arguments

version	A string defining the version to compare to, like '2.0.0' or '1.9.1'.

Value

-1 if R is older, 0 if equal, 1 if newer. Take care: if you specify version as "2.11", and R is version "2.11.0", then the function will return 1 (newer)!

See Also

compareVersion(), R.version()

Examples

```
compare_r_version("2.11.0") # Note that we strongly advise to use R > 2.11.0!
```

completion *Get a completion list for a R code fragment*

Description

Returns names of objects/arguments/namespaces matching a code fragment.

Usage

```
completion(code, pos = nchar(code), min.length = 2, print = FALSE,
  types = c("default", "scintilla"), addition = FALSE, sort = TRUE,
  what = c("arguments", "functions", "packages"), description = FALSE,
  max.fun = 100, skip.used.args = TRUE, sep = "\n", field.sep = "\t")
```

Arguments

code	A partial R code to be completed.
pos	The position of the cursor in this code.
min.length	The minimal length in characters of code required before the completion list is calculated.
print	Logical, print result and return invisibly. See details.
types	A named list giving names of types. Set to NA to give only names. See details.
addition	Should only addition string be returned?
sort	Do we sort the list of completions alphabetically?
what	What are we looking for? Allow to restrict search for faster calculation.
description	Do we describe items in the completion list (could be slow)?
max.fun	In the case where we describe items, the maximum number of functions to process (if longer, no description is returned for function) because it can be very slow otherwise.
skip.used.args	Logical, if completion is within function arguments, should the already used named arguments be omitted?
sep	The separator to use between returned items.
field.sep	Character string to separate fields for each entry.

Details

The completion list is context-dependent, and it is calculated as if the code was entered at the command line.

If the code ends with $ or [[, then the function look for items in a list or data.frame whose name is the last identifier.

If the code ends with @, then the function look for slots of the corresponding S4 object.

If the code ends with ::, then it looks for objects in a namespace.

If the code ends with a partial identifier name, the function returns all matching keywords visible from .GlobalEnv.

If the code is empty or parses into an empty last token, the list of objects currently in the global environment is returned.

Value

If types == NA and description = FALSE, a character vector giving the completions, otherwise a data frame with two columns: 'completion', and 'type' when description = FALSE, or with four columns: 'completion', 'type', 'desc' and 'context' when description = TRUE.
Attributes:
attr(, "token") - a completed token.
attr(, "triggerPos") - number of already typed characters.
attr(, "fguess") - name of guessed function.
'attr(, "isFirstArg")" - is this a first argument?

Note

Take care: depending on the context, the completion list could be incorrect (but it should work for code entered at the command line). For instance, inside a function call, the context is very different, and arguments and local variables should be returned instead. This may be implemented in the future, but for now, we focus on completion that should be most useful for novice useRs that are using R expressions entered one after the other at the R console or in a script (and considering the script is run or sourced line after line in R).

There are other situations where the completion can be calculated, see the help of `rc.settings()`.

If `print == TRUE`, results are returned invisibly, and printed in a form: triggerPos<newline>completions separated by `sep`.

If `types` are supplied, a completion will consist of name and type, separated by `type.sep`. `types` may me a vector of length 5, giving the type codes for "function", "variable", "environment", "argument" and "keyword". If `types == "default"`, above type names are given; `types == "scintilla"` will give numeric codes that can be used with "scintilla.autoCShow" function (e.g., with the SciViews-K Komodo Edit plugin).

Author(s)

Philippe Grosjean phgrosjean@sciviews.org & Kamil Barton kamil.barton@uni-wuerzburg.de

See Also

`rc.settings()`

Examples

```
# A data frame
data(iris)
completion("item <- iris$")
completion("item <- iris[[")

# An S4 object
setClass("track", representation(x = "numeric", y = "numeric"))
t1 <- new("track", x = 1:20, y = (1:20)^2)
completion("item2 <- t1@")

# A namespace
completion("utils::", description = TRUE)

# A partial identifier
completion("item3 <- va", description = TRUE)

# Otherwise, a list with the content of .GlobalEnv
completion("item4 <- ")

# TODO: directory and filename completion!
rm(iris, t1)
```

def *Define a vector of a given mode and length (possibly filling it with default values)*

Description

This function makes sure that a vector of a given mode and length is returned. If the value provided is NULL, or empty, the default value is used instead. If `length.out = NULL`, the length of the vector is not constrained, otherwise, it is fixed (possibly cutting or recycling `value`).

Usage

```
def(value, default = "", mode = "character", length.out = NULL)
```

Arguments

`value`	The value to pass with default.
`default`	The default value to use, in case of NULL, or `length(value) == 0`.
`mode`	The mode of the resulting object: 'character', 'logical', 'numeric' (and, if you want to be more precise: 'double', 'integer' or 'single') or 'complex'. Although not being a mode by itself, you can also specify 'factor' to make sure the result is a factor (thus, of mode 'numeric', storage mode 'integer', with a levels attribute). Other modes are ignored, and `value` is NOT coerced (silently) in this case, i.e., if you don't want to force coercion of the resulting object, use anything else.
`length.out`	The desired length of the returned vector; use `length.out = NULL` (default) if you don't want to change the length of the vector.

Value

A vector of given mode and length, with either `value` or `default`.

See Also

`mode()`, `rep()`, `temp_env()`

Examples

```
def(1:3, length.out = 5)                    # Convert into character and recycle
def(0:2, mode = "logical")                  # Numbers to logical
def(c("TRUE", "FALSE"), mode = "logical")   # Text to logical
def(NULL, "default text")                   # Default value used
def(character(0), "default text")           # Idem
def(NA, 10, mode = "numeric", length.out = 2) # Vector of two numbers
```

describe_function *Get textual help on function or function arguments, or get a call tip*

Description

Textual help on functions or their arguments is extracted for text online help for a given function. By default, all arguments from the online help are returned for `describe_args()`. If the file contains help for several functions, one probably gets also some irrelevant information. Use of 'args' to limit result is strongly encouraged. `args_tip()` provides a human-readable textual description of function arguments in a better way than `args()` does. It is primarily intended for code tips in GUIs. `call_tip()` has a similar purpose to show how some code could be completed.

Usage

```
describe_function(fun, package, lib.loc = NULL)

describe_args(fun, args = NULL, package = NULL, lib.loc = NULL)

args_tip(name, only.args = FALSE, width = getOption("width"))

call_tip(code, only.args = FALSE, location = FALSE, description = FALSE,
  methods = FALSE, width = getOption("width"))

descFun(fun, package, lib.loc = NULL)

descArgs(fun, args = NULL, package = NULL, lib.loc = NULL)

argsTip(name, only.args = FALSE, width = getOption("width"))

callTip(code, only.args = FALSE, location = FALSE, description = FALSE,
  methods = FALSE, width = getOption("width"))
```

Arguments

fun	A character string with the name of a function (several functions accepted too for `describe_function()`.
package	A character string with the name of the package that contains `fun`, or `NULL` for searching in all loaded packages.
lib.loc	A character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
args	Either `NULL` (by default) to return the description of all arguments from the corresponding man page, or a character vector with names of the arguments to search for.
name	A string with the name of a function.

only.args	Do we return only arguments of the function (arg1, arg2 = TRUE, ...), or the full call, like (myfun(arg1, arg2 = TRUE, ...)).
width	Reformat the tip to fit to fit in that width, except if width = NULL.
code	A fraction of R code ending with the name of a function, eventually followed by '('.
location	If TRUE then the location (in which package the function resides) is appended to the calltip between square brackets.
description	If TRUE then a short description of the function is added to the call_tip (in fact, the title of the corresponding help page, if it exists).
methods	If TRUE then a short message indicating if this is a generic function and that lists, in this case, available methods.

Value

A string with the description of the function or of its arguments, or the calling syntax of the function, plus additional information depending on the flags used. If the man page is not found, a vector of empty strings is returned. Empty strings are also returned for arguments that are not found in the man page.

Note

args_tip() is supposed to display S3 and S4 methods, and primitives adequately,... but this is not implemented yet in the current version! For call_tip(), the use of methods = TRUE slows down the execution of the function, especially for generic functions that have many methods like print() or summary().

See Also

completion(), args(), argsAnywhere()

Examples

```
describe_function("ls", "base")
describe_function("library", "base")
describe_function("descFun", "svMisc")
describe_function("descArgs")

describe_args("ls")
describe_args("library", args = c("package", "pos"))

args_tip("ls")

call_tip("myvar <- lm(")
```

`file_edit` *Invoke an external text editor for a file*

Description

Edit a text file using an external editor. Possibly wait for the end of the program and care about creating the file (from a template) if it does not exists yet.

Usage

```
file_edit(..., title = files, editor = getOption("fileEditor"),
  file.encoding = "", template = NULL, replace = FALSE, wait = FALSE)

fileEdit(..., title = files, editor = getOption("fileEditor"),
  file.encoding = "", template = NULL, replace = FALSE, wait = FALSE)
```

Arguments

`...`	Path to one or more files to edit.
`title`	The title of the editor window (not honoured by all editors, most external editors only display the file name or path).
`editor`	Editor to use. Either the name of the program, or a string containing the command to run, using %s as replacement tag where to place the filename in the command, or a function with 'file', 'title' and 'wait' arguments to delegate process of the files.
`file.encoding`	Encoding of the files. If `""` or `native.enc`, the files are considered as being already in the right encoding.
`template`	One or more files to use as template if files must be created. If `NULL`, an empty file is created. This argument is recycled for all files to edit.
`replace`	Force replacement of files if `template=` is not null.
`wait`	Wait for edition to complete. If more than one file is edited, the program waits sequentially for each file to be edited in turn (with a message in the R console).

Value

The function returns `TRUE` if it was able to edit the files or `FALSE` otherwise, invisibly. Encountered errors are reported as warnings.

Note

The default editor program, or the command to run is in the `fileEditor` option (use `getOption("fileEditor")` to retrieve it, and `options(fileEditor = "<my_own_editor>")` to change it). Default values are determined automatically.

On Unixes, "gedit", "kate" and "vi" are looked for in that order. Note that there is a gedit plugin to submit code directly to R: http://rgedit.sourceforge.net/. Since, gedit natively supports a lot

of different syntax highlighting, including R, and is lightweight but feature rich, it is recommended as default text editor for `file_edit()` on Unixes. If JGR is run and the editor is "vi" or "internal", then the internal JGR editor is used, otherwise, the provided editor is choosen.

On MacOS, if the "bbedit" program exists, it is used (it is the command line program installed by BBEdit, see http://www.barebones.com/products/, a much more capables text editor than the default TextEdit program), otherwise, the default text editor used by MacOS is choosen (default usually to TextEdit). BBEdit can be configured to highlight and submit R code.It features also several tools that makes it a much better choice than TextEdit for `file_edit()` on MacOS. Specify "bbedit" to force using it. The default value is "textedit", the MacOS default text editor, but on R.app, and with `wait = FALSE`, the internal R.app editor is used instead in that case. If RStudio or JGR is run, and the editor is "textedit", "internal" or "vi", then, the RStuiod or JGR internal editor is used instead. If `wait = TRUE` with an RStudio editor, it is enough to switch to another editor to continue.

On Windows, if Notepad++ is installed in its default location, it is used, otherwise, the default "notepad" is used in Rterm and the internal editors are choosen for Rgui. Notepad++ is a free text editor that is much better suited to edit code or text files that the default Windows' notepad application, in particular because it can handle various line end types (Unix, Mac or Windows) and encodings. It also supports syntax highlighting, code completion and much more. So, it is strongly recommended to install it (see http://notepad-plus-plus.org/) and use it with `file-edit()`. There is also a plugin to submit code to R directly from Notepad++: http://sourceforge.net/projects/npptor/.

Of course, you can use your own text editor, just indicate it in the `fileEditor` option. Note, however, that you should use only lighweight and fast starting programs. Also, for the `wait = TRUE` argument of `file_edit()`, you must check that R waits for the editor to be closed before further processing code. In some cases, a little command line program is used to start a larger application (like for Komodo Edit/IDE), or the program delegates to an existing instances and exits immediatelly, even if the file is still edited. Such editors are not recommended at all for `file_edit()`.

If you want to use files that are compatibles between all platforms supported by R itself, you should think about using ASCII encoding as much as possible and the Windows style of line-ending. That way, you ensure that all the default editors will handle those files correctly, including the broken default editor on Windows, notepad, which does not understand at all MacOS or Unix line ending characters!

See Also

`system_file()`, `file.path()`, `file.edit()`

Examples

```
## Not run:
# Create a template file in the tempdir...
template <- tempfile("template", fileext = ".txt")
cat("Example template file to be used with file_edit()", file = template)

# ... and edit a new file, starting from that template:
new_file <- tempfile("test", fileext = ".txt")
file_edit(new_file, template = template, wait = TRUE)
```

```
message("Your file contains:")
readLines(new_file)

# Eliminate both the file and template
unlink(new_file)
unlink(template)

## End(Not run)
```

---

gui_cmd *Execute a command in the GUI client*

Description

This function is not intended to be used at the command line (except for debugging purposes). It executes a command string to a (compatible) GUI client.

Usage

```
gui_cmd(command, ...)

gui_load(...)

gui_source(...)

gui_save(...)

gui_import(...)

gui_export(...)

gui_report(...)

gui_setwd(...)

guiCmd(command, ...)

guiLoad(...)

guiSource(...)

guiSave(...)

guiImport(...)

guiExport(...)

guiReport(...)
```

```
guiSetwd(...)
```

Arguments

command	The command string to execute in the GUI client.
...	Parameters provided for the command to execute in the GUI client.

Details

You must define a function `.guiCmd()` in the `SciViews:TempEnv` environment that will take first argument as the name of the command to execute (like `source`, `save`, `import`, etc.), and ... with arguments to the command to send. Depending on your GUI, you should have code that delegates the GUI elements (ex: display a dialog asking for a .Rdata file to source) and then, execute the command in R with the selected file as attribute.

Value

The result of the command if it succeed, or `NULL` if the command cannot be run (i.e., `.guiCmd()` is not defined in `SciViews:TempEnv`).

See Also

`get_temp()`

Install *An easy package installation function that pairs with* `package()`

Description

This is similar to `install.packages()`, except it takes by default the list of packages from `.packages_to_install` in `SciViews:TempEnv`. That list is populated automatically by unfructuous calls to `package()`, so that just a call to `install()` without arguments is generaly sufficient.

Usage

```
Install(pkgs = get_temp(".packages_to_install"), ..., ask = TRUE)
```

Arguments

pkgs	The list of packages to install (character vector). If missing, the list is read from `packages_to_install`, which is cleared on success.
...	Further arguments passed to `install.packages()`.
ask	If `TRUE` and `pkgs` is missing, ask first to install the packages.

Value

Returns TRUE in case of success, FALSE otherwise. The function is invoked for its side-effect of installing R packages.

See Also

package()

---

is_help *Check for the existence of an help file, or some context*

Description

For is_help(), determine if 'topic' has a help file and example to run. For is_win() and is_mac(), determine if the platform is Windows or MacOS. For is_aqua(), is the R UI is AQUA, the standard R GUI under Macintosh? For is_rgui(), determine if the default Rgui under Windows is in use, and with is_sdi() in this case, you can check if it is in SDI (single-document interface) *versus* MDI (multi-document interface, by default). is_rstudio() and is_rstudio_server() check if R is run under RStudio (server), and is_jgr() indicate if the R GUI is JGR.

Usage

```
is_help(topic, package = NULL, lib.loc = NULL)

is_win()

is_rgui()

is_sdi()

is_mac()

is_aqua()

is_rstudio()

is_rstudio_desktop()

is_rstudio_server()

is_jgr()

isHelp(topic, package = NULL, lib.loc = NULL)

isWin()

isRgui()
```

```
isSDI()

isMac()

isAqua()

isJGR()
```

Arguments

topic	Name or literal character string: the online help topic to look for.
package	A character vector giving the package names to look into for help or example code, or NULL. By default, all packages in the search path are used.
lib.loc	A character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.

Value

All these functions return either TRUE or FALSE depending on the tested item, except for `is_help()`, which returns a logical vector with two elements. The first one indicating if there is a help file, and the second one indicating if there are examples associated with this help file.

Note

The code of `is_help()` is largely inspired from the first part of `example()`.

Under **Rgui**, to switch fro MDI to SDI more, go to the menu entry 'Edit' -> 'GUI preferences' to change the Rgui mode, or start Rgui with the '–SDI' argument line parameter. Under another platform than Windows or if it is not Rgui, then `is_sdi()` always returns FALSE.‘

See Also

`example()`, `help()`, `capabilities()`

Examples

```
is_help("help") # Help and example
is_help("Rtangle") # Help but no example
is_help("notopic") # No help or example

is_win()
is_mac()

is_aqua()
is_rgui()
is_sdi()
is_rstudio()
is_rstudio_desktop()
is_rstudio_server()
```

```
is_jgr()
```

`list_methods` *List all methods associated with a generic function or a class, or all types associated with a method*

Description

List all S3 and/or S4 methods for a generic function or for a class. List all types for a method; types are variants for a given method defined in a way it is easy to add other variants dynamically (on the contrary to a usual `type =` or `which =` argument, like in `plot.ts()` or `plot.lm()`, respectively.

Usage

```
list_methods(f = character(), class = NULL, S3 = TRUE, S4 = TRUE,
  mixed = TRUE, filter = getOption("svGUI.methods"))

list_types(method, class = "default", strict = FALSE)

listMethods(f = character(), class = NULL, S3 = TRUE, S4 = TRUE,
  mixed = TRUE, filter = getOption("svGUI.methods"))

listTypes(method, class = "default", strict = FALSE)
```

Arguments

`f`	The name of the generic function (character string), used only when `class = NULL`.
`class`	The name of a class.
`S3`	If `TRUE`, list of S3 methods.
`S4`	If `TRUE`, list of S4 methods.
`mixed`	If `TRUE`, S3 and S4 methods are mixed together in a character vector, otherwise, S3 and S4 methods are reported separately in a list.
`filter`	A list of methods to consider when listing class methods. Only classes in this list that are defined for the class are returned. Store the list of methods you want in the options `"svGUI.methods"`. The package proposes a reasonable starting point on loading if this option is not defined yet.
`method`	The method name.
`strict`	Do we list only types for the class (`TRUE`), or all possible types, including for inherited objects, and default ones `FALSE`, by default)?

Value

For `list_methods()`, if `mixed = TRUE`, a list with components:

- `S3` The S3 methods for the generic function or the class, or `character(0)` if none
- `S4` The S4 methods for the generic function or the class, or `character(0)` if none.

Otherwise, a character vector with the requested methods.

For `list_types()`, a vector with character strings with methods' type names.

Note

`list_types()` is only useful for special generic functions with type argument like `view`, `copy` or `export`. These functions offer a mechanism to easily add custom types, and the present function list them. For S3 objects a type is simply a function whose name is : <method>_<type>.<class>. So, adding new <type>s for <method> is very easy to implement.

See Also

`obj_menu()`

Examples

```
# Generic functions
list_methods("t.test")               # S3
list_methods("show", mixed = FALSE) # S4
list_methods("ls") # None, not a generic function!

# Classes
# Only the following methods are considered
getOption("gui.methods")
list_methods(class = "data.frame")
list_methods(class = "lm")

# List method types
list_types("view")  # All default view types currently defined
list_types("view", "data.frame")
list_types("view", "data.frame", TRUE) # None, except if you defined custom views!
```

---

obj_browse *Functions to implement an object browser*

Description

These functions provide features required to implement a complete object browser in a GUI client.

Usage

```
obj_browse(id = "default", envir = .GlobalEnv, all.names = NULL,
  pattern = NULL, group = NULL, sep = "\t", path = NULL,
  regenerate = FALSE)

obj_clear(id = "default")

obj_dir()

obj_info(id = "default", envir = .GlobalEnv, object = "", path = NULL)

obj_list(id = "default", envir = .GlobalEnv, object = NULL,
  all.names = FALSE, pattern = "", group = "", all.info = FALSE,
  sep = "\t", path = NULL, compare = TRUE, ...)

write.objList(x, path, sep = "\t", ...)

## S3 method for class 'objList'
print(x, sep = NA, eol = "\n", header = !attr(x,
  "all.info"), raw.output = !is.na(sep), ...)

obj_search(sep = "\t", path = NULL, compare = TRUE)

obj_menu(id = "default", envir = .GlobalEnv, objects = "", sep = "\t",
  path = NULL)

objBrowse(id = "default", envir = .GlobalEnv, all.names = NULL,
  pattern = NULL, group = NULL, sep = "\t", path = NULL,
  regenerate = FALSE)

objClear(id = "default")

objDir()

objInfo(id = "default", envir = .GlobalEnv, object = "", path = NULL)

objList(id = "default", envir = .GlobalEnv, object = NULL,
  all.names = FALSE, pattern = "", group = "", all.info = FALSE,
  sep = "\t", path = NULL, compare = TRUE, ...)

objSearch(sep = "\t", path = NULL, compare = TRUE)

objMenu(id = "default", envir = .GlobalEnv, objects = "", sep = "\t",
  path = NULL)
```

Arguments

id	The id of the object browser (you can run several ones concurrently, providing you give them different ids).
envir	An environment, or the name of the environment, or the position in the search() path.
all.names	Do we display all names (including hidden variables starting with '.')?
pattern	A pattern to match for selecting variables.
group	A group to filter.
sep	Separator to use between items (if path is not NULL).
path	The path where to write a temporary file with the requested information. Set to NULL (default) if you don't pass this data to your GUI client by mean of a file.
regenerate	Do we force to regenerate the information?
object	Name of the object selected in the object browser, components/arguments of which should be listed.
all.info	Do we return all the information (envir as first column or not (by default).
compare	If TRUE, result is compared with last cached value and the client is updated only if something changed.
...	Further arguments, passed to write.table().
x	Object returned by obj_list().
eol	Separator to use between object entries, default is to list each item in a separate line.
header	If TRUE, two-line header is printed, of the form: Environment = environment name Object = object name Default is not to print header if all.info == TRUE.
raw.output	If TRUE, a compact, better suited for parsing output is produced.
objects	A list with selected items in the object browser.

Details

obj_browse() does the horsework. obj_dir() gets the temporary directory where exchange files with the GUI client are stored, in case you exchange data through files. You can use a better way to communicate with your GUI (you have to provide your code) and disable writing to files by using path = NULL.

obj_list() lists objects in a given environment, elements of a recursive object or function argument.

obj_search() lists the search path.

obj_clear() clears any reference to a given object browser.

obj_info() computes a tooltip info for a given object.

obj_menu()‘ computes a context menu for selected object(s) in the object explorer managed by the GUI client.

print.objList() print method for objList objects.

Value

Depending on the function, a list, a string, a reference to an external, temporary file or TRUE in case of success or FALSE otherwise is returned invisibly.

Author(s)

Philippe Grosjean phgrosjean@sciviews.org & Kamil Barton kamil.barton@uni-wuerzburg.de

See Also

completion(), call_tip()

Examples

```
# Create various context menus
data(iris)
(obj_info(object = "iris"))
data(trees)
# For one object
(obj_menu(objects = "iris"))
# For multiple objects
(obj_menu(objects = c("iris", "trees")))
# For inexistant object (return "")
(obj_info(object = "noobject"))
(obj_menu(objects = "noobject"))
rm(iris, trees)

# For environments
(obj_info(envir = ".GlobalEnv"))
(obj_menu(envir = ".GlobalEnv"))
(obj_info(envir = "SciViews:TempEnv"))
(obj_menu(envir = "SciViews:TempEnv"))
(obj_info(envir = "package:datasets"))
(obj_menu(envir = "package:datasets"))
# For an environment that does not exist on the search path (return "")
(obj_info(envir = "noenvir"))
(obj_menu(envir = "noenvir"))
```

---

package *A (possibly) very silent and multi-package library()/require() function*

Description

This function loads one or several R packages as silently as possible (with warn/message = FALSE) and it returns TRUE only if all packages are loaded successfully. If at least one loading fails, a short message is printed, by default. For all packages that were not found, an entry is recorded in .packages_to_install in SciViews:TempEnv, and that list can be automatically used by Install().

Usage

```
package(..., stop = TRUE, message = stop, warn.conflicts = message,
  pos = 2L, lib.loc = NULL, verbose = getOption("verbose"))
```

Arguments

`...`	The name of one or several R packages to load (character strings).
`stop`	If `TRUE`, issue an error in case the package(s) cannot be loaded.
`message`	Do we display introductory message of the package? If a package displays such a message, there is often a good reason. So, it is **not** a good idea to disable it in *interactive* sessions. However, in other contexts, like in non-interactive use, inside an R Markdown document, etc., it is more convenient not to display it.
`warn.conflicts`	As for `library()`: "logical. If TRUE, warnings are printed about conflicts from attaching the new package. A conflict is a function masking a function, or a non-function masking a non-function.
`pos`	As for `library()`: "the position on the search list at which to attach the loaded namespace. Can also be the name of a position on the current search list as given by `search()`". Only one position can be provided here, even if several packages, and they will be all inserted one after the other at the given position.
`lib.loc`	As for `library()`: "a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known to `.libPaths()`. Non-existent library trees are silently ignored".
`verbose`	A logical indicating if additional diagnostic messages are printed.

Value

`TRUE` if all packages are loaded correctly, `FALSE` otherwise, with a `details` attribute indicating which package was loaded or not.

Note

This function is designed to concisely and possibly quietly (with `warn = FALSE`) load packages and attach them to the search path. Also, on the contrary to `library()`, or `require()`, it is **not** possible to use unquoted names of the packages. This is cleaner, and avoids the contrived work-around to pass name(s) of packages as a variable with an arguments `character.only = TRUE`!

If several packages are provided, they are loaded and attached in reverse order, so that the order in the search path is the same one as the order in the provided vector.

The `library(help = ...)` version is not implemented here.

See Also

`require()`, `library()`, `Install()`

Examples

```
# This should work...
if (package('tools', 'methods', stop = FALSE)) message("Fine!")
# ... but this not (note that there are no details here!)
if (!package('tools', 'badname', stop = FALSE)) message("Not fine!")
## Not run:
# Get an error
package('badname')

## End(Not run)
```

`parse_text` *Parse a character string as if it was a command entered at the command line*

Description

Parse R instructions provided as a string and return the expression if it is correct, or an object of class 'try-error' if it is an incorrect code, or NA if the (last) instruction is incomplete.

Usage

```
parse_text(text, firstline = 1, srcfilename = NULL, encoding = "unknown")

parseText(text, firstline = 1, srcfilename = NULL, encoding = "unknown")
```

Arguments

`text`	The character string vector to parse into an R expression.
`firstline`	The index of first line being parsed in the file. If this is larger than 1, empty lines are added in front of `text` in order to match the correct position in the file.
`srcfilename`	A character string with the name of the source file.
`encoding`	Encoding of 'text", as in `parse()`.

Value

Returns an expression with the parsed code or NA if the last instruction is correct but incomplete, or an object of class'try-error' with the error message if the code is incorrect.

Note

On the contrary to `parse()`, `parse_text()` recovers from incorrect code and also detects incomplete code. It is also easier to use in case you pass a character string to it, because you don't have to name the argument explicitly (`text = ...`).

See Also

`parse()`, `capture_all()`

Examples

```
parse_text("1 + 1")
parse_text("1 + 1; log(10)")
parse_text(c("1 + 1", "log(10)"))

# Incomplete instruction
parse_text("log(")

# Incomplete strings
parse_text("text <- \"some string")
parse_text("text <- 'some string")

# Incomplete names (don't write backtick quoted names on several lines!)
# ...but just in case
parse_text("`myvar")

# Incorrect expression
parse_text("log)")
```

---

pkgman_describe *Functions to manage R side of the SciViews R package manager*

Description

These functions should not be used directly by the end-user. They implement the R-side code for the SciViews R package manager.

Usage

```
pkgman_describe(pkgname, print.it = TRUE)

pkgman_get_mirrors()

pkgman_get_available(page = "next", pattern = "", n = 50,
  keep = c("Package", "Version", "InstalledVersion", "Status"),
  reload = FALSE, sep = ";", eol = "\t\n")

pkgman_get_installed(sep = ";", eol = "\t\n")

pkgman_set_cran_mirror(url)

pkgman_install(pkgs, install.deps = FALSE, ask = TRUE)

pkgman_remove(pkgname)

pkgman_load(pkgname)

pkgman_detach(pkgname)
```

```
pkgManDescribe(pkgname, print.it = TRUE)

pkgManGetMirrors()

pkgManGetAvailable(page = "next", pattern = "", n = 50,
  keep = c("Package", "Version", "InstalledVersion", "Status"),
  reload = FALSE, sep = ";", eol = "\t\n")

pkgManGetInstalled(sep = ";", eol = "\t\n")

pkgManSetCRANMirror(url)

pkgManInstall(pkgs, install.deps = FALSE, ask = TRUE)

pkgManRemove(pkgname)

pkgManLoad(pkgname)

pkgManDetach(pkgname)
```

Arguments

pkgname	The name of one R package (character string).
print.it	Should the result be printed?
page	Which page to get?
pattern	Selection pattern.
n	The number of items to retrieve.
keep	The columns to keep in the resulting data frame.
reload	Do we force reload of the data and ignore cache version?
sep	Field separator to use.
eol	End-of-line sequence to use.
url	The URL to use for the current CRAN mirror.
pkgs	A list of packages to install.
install.deps	Do we also install dependencies?
ask	Do we prompt the user for package installation?

Value

These functions return data that is intended to be used by the SciViews R package manager.

Author(s)

Kamil Barton kamil.barton@uni-wuerzburg.de

See Also

package()

`progress` *Display progression of a long calculation at the R console and/or in a GUI*

Description

Display progression level of a long-running task in the console. Two mode can be used: either percent of achievement (55%), or the number of items or steps done on a total (1 file on 10 done...). This is displayed either through a message, or through a text-based "progression bar" on the console, or a true progression bar widget in a GUI.

Usage

```
progress(value, max.value = NULL, progress.bar = FALSE, char = "|",
  init = (value == 0), console = TRUE, gui = TRUE)
```

Arguments

`value`	Current value of the progression (use a value higher than `max.value` to dismiss the progression indication automatically.
`max.value`	Maximum value to be achieved.
`progress.bar`	Should we display a progression bar on the console? If `FALSE`, a temporary message is used.
`char`	The character to use to fill the progress bar in the console. not used for the alternate display, or for GUI display of progression.
`init`	Do we have to initialize the progress bar? It is usually done the first time the function is used, and the default value `init = (value == 0)` is correct most of the time. You must specify `init = TRUE` in two cases: (1) if the first value to display is different from zero, and (2) if your code issues some text on screen during progression display. Hence, you must force redraw of the progression bar.
`console`	Do we display progression on the console?
`gui`	Do we display progression in a dialog box, or any other GUI widget? See "details" and "examples" hereunder to know how to implement your own GUI progression indicator.

Details

The function progress() proposes different styles of progression indicators than the standard txtProgressBar() in package 'utils'.

The function uses backspace (\8) to erase characters at the console.

With gui = TRUE, it looks for all functions defined in the .progress list located in the SciViews:TempEnv environment. Each function is executed in turn with following call: the_gui_function(value, max.value). You are responsible to create the_gui_function() and to add it as an element in the .progress list. See also example (5) hereunder.

If your GUI display of the progression offers the possibility to stop calculation (for instance, using a 'Cancel' button), you are responsible to pass this info to your code doing the long calculation and to stop it there. Example (5) shows how to do this.

Value

This function returns NULL invisibly. It is invoked for its side effects.

See Also

batch(), txtProgressBar()

Examples

```
# 1) A simple progress indicator in percent
for (i in 0:101) {
  progress(i)
  Sys.sleep(0.01)
  if (i == 101) message("Done!")
}

# 2) A progress indicator with 'x on y'
for (i in 0:31) {
  progress(i, 30)
  Sys.sleep(0.02)
  if (i == 31) message("Done!")
}

# 3) A progress bar in percent
for (i in 0:101) {
  progress(i, progress.bar = TRUE)
  Sys.sleep(0.01)
  if (i == 101) message("Done!")
}

# 4) A progress indicator with 'x on y'
for (i in 0:21) {
  progress(i, 20, progress.bar = TRUE)
  Sys.sleep(0.03)
  if (i == 21) message("Done!")
}
```

```
# 5) A progression dialog box with Tcl/Tk
## Not run:
if (require(tcltk)) {
  gui_progress <- function(value, max.value) {
    # Do we need to destroy the progression dialog box?
    if (value > max.value) {
      try(tkdestroy(get_temp("gui_progress_window")), silent = TRUE)
      delete_temp(c("gui_progress_state", "gui_progress_window",
        "gui_progress_cancel"))
      return(invisible(FALSE))
    } else if (exists_temp("gui_progress_window") &&
      !inherits(try(tkwm.deiconify(tt <- get_temp("gui_progress_window")),
        silent = TRUE), "try-error")) {
      # The progression dialog box exists
      # Focus on it and change progress value
      tkfocus(tt)
      state <- get_temp("gui_progress_state")
      tclvalue(state) <- value
    } else {
      # The progression dialog box must be (re)created
      # First, make sure there is no remaining "gui_progress_cancel"
      delete_temp("gui_progress_cancel")
      # Create a Tcl variable to hold current progression state
      state <- tclVar(value)
      assign_temp("gui_progress_state", state)
      # Create the progression dialog box
      tt <- tktoplevel()
      assign_temp("gui_progress_window", tt)
      tktitle(tt) <- "Waiting..."
      sc <- tkscale(tt, orient = "h", state = "disabled", to = max.value,
        label = "Progress:", length = 200, variable = state)
      tkpack(sc)
      but <- tkbutton(tt, text = "Cancel", command = function() {
        # Set a flag telling to stop running calculation
        assign_temp("gui_progress_cancel", TRUE) # Content is not important!
        tkdestroy(tt)
      })
      tkpack(but)
    }
    invisible(TRUE)
  }
  # Register it as function to use in progress()
  change_temp(".progress", "gui_progress", gui_progress,
    replace.existing = TRUE)
  rm(gui_progress) # Don't need this any more
  # Test it...
  for (i in 0:101) {
    progress(i) # Could also set console = FALSE for using the GUI only
    Sys.sleep(0.05)
    # The code to stop long calc when user presses "Cancel"
    if (exists_temp("gui_progress_cancel")) {
      progress(101, console = FALSE) # Make sure to clean up everything
      break
```

```
    }
    if (i == 101) message("Done!")
  }
  # Unregister the GUI for progress
  change_temp(".progress", "gui_progress", NULL)
}

## End(Not run)
```

---

search_web *Search web documents about R and R functions*

Description

Retrieve web documents, or search with Google for `what` string.

Usage

```
search_web(what, type = c("R", "google"), browse = TRUE, msg = browse,
  ...)

helpSearchWeb(what, type = c("R", "google"), browse = TRUE, msg = browse,
  ...)
```

Arguments

`what`	The string(s) to search. In case of several strings, or several words, any of these words are searched.
`type`	The search engine, or location to use.
`browse`	Do we actually show the page in the Web browser? If `type = "R"`, this argument is ignored and the result is always displayed in the Web browser.
`msg`	Do we issue a message indicating that a page should be displayed shortly in the Web browser? If `type = "R"`, this argument is ignored and a message is always displayed.
`...`	Further arguments to format the result page in case of `type = "R"`. These are the same arguments as for `RSiteSearch()`.

Value

Returns the URL used invisibly (invoked for its side effect of opening the Web browser with the search result, when `browse = TRUE`).

Note

The `RSiteSearch()` function in the 'utils' package is used when `type = "R"`.

See Also

RSiteSearch(), help.search()

Examples

```
## Not run:
search_web("volatility")                    # R site search, by default
search_web("volatility", type = "google")   # Google search

## End(Not run)
```

---

source_clipboard *Source code from the clipboard*

Description

This function reads R code from the clipboard, and then source it. Clipboard is managed correctly depending on the OS (Windows, MacOS, or *nix)

Usage

```
source_clipboard(primary = TRUE, ...)

sourceClipboard(primary = TRUE, ...)
```

Arguments

primary	Only valid on *nix: read the primary (or secondary) clipboard.
...	Further parameters passed to source().

Value

Same result as source().

See Also

source(), file()

subsettable *Define a function as being 'subsettable' using $ operator*

Description

In case a textual argument allows for selecting the result, for instance, if `plot()` allows for several charts that you can choose with a `type=` or `which=`, making the function 'subsettable' also allows to indicate `fun$variant()`. See examples.

Usage

```
## S3 method for class 'subsettable_type'
x$name

## S3 method for class 'subsettable_which'
x$name
```

Arguments

x	A `subsettable_type` function.
name	The value to use for the `type=` argument.

Examples

```
foo <- structure(function(x, type = c("histogram", "boxplot"), ...) {
  type <- match.arg(type, c("histogram", "boxplot"))
  switch(type,
    histogram = hist(x, ...),
    boxplot = boxplot(x, ...),
    stop("unknow type")
  )
}, class = c("function", "subsettable_type"))
foo

# This function can be used as usual:
foo(rnorm(50), type = "histogram")
# ... but also this way:
foo$histogram(rnorm(50))
foo$boxplot(rnorm(50))
```

`system_file` *Get a system file or directory*

Description

Get system files or directories, in R subdirectories, in package subdirectories, or elsewhere on the disk (including executables that are accessible on the search path).

Usage

```
system_file(..., exec = FALSE, package = NULL, lib.loc = NULL)

system_dir(..., exec = FALSE, package = NULL, lib.loc = NULL)

systemFile(..., exec = FALSE, package = NULL, lib.loc = NULL)

systemDir(..., exec = FALSE, package = NULL, lib.loc = NULL)
```

Arguments

`...`	One or several executables if `exec = TRUE`, or subpath to a file or dir in a package directory if `package != NULL`, or a list of paths and subpaths for testing the existence of a file on disk, or a list of directory components to retrieve in 'temp', 'sysTemp', 'user', 'home', 'bin', 'doc', 'etc' and/or 'share' to retrieve special system directories.
`exec`	If `TRUE` (default) search for executables on the search path. It superseedes all other arguments.
`package`	The name of one package to look for files or subdirs in its main directory (use `exec = FALSE` to search inside package dirs).
`lib.loc`	A character vector with path names of R libraries or `NULL` (search all currently known libraries in this case).

Value

A string with the path to the directories or files, or `""` if they are not found, or of the wrong type (a dir for `system_file()` or or a file for `system_dir()`).

Note

These function aggregate the features of several R functions in package base: `system.file()`, `R.home()`, `tempdir()`, `Sys.which()`, and aims to provide a unified and convenient single interface to all of them. We make sure also to check that returned components are respectively directories and files for `system_dir()` and `system_file()`.

See Also

`file_edit()`, `file.path()`, `file.exists()`

Examples

```
system_file("INDEX", package = "base")
system_file("help", "AnIndex", package = "splines")
system_file(package = "base")  # This is a dir, not a file!
system_file("zip", exec = TRUE)
system_file("ftp", "ping", "zip", "nonexistingexe", exec = TRUE)
system_dir("temp")             # The R temporary directory
system_dir("sysTemp")          # The system temporary directory
system_dir("user")             # The user directory
system_dir("home", "bin", "doc", "etc", "share")  # Various R dirs
system_dir("zip", exec = TRUE) # Look for the dir of an executable
system_dir("ftp", "ping", "zip", "nonexistingexe", exec = TRUE)
system_dir(package = "base")   # The root of the 'base' package
system_dir(package = "stats")  # The root of package 'stats'
system_dir("INDEX", package = "stats") # This is a file, not a dir!
system_dir("help", package = "splines")
```

`temp_env` *Get an environment dedicated to temporary variables (and create it if needed)*

Description

Create and manage a temporary environment `SciViews:TempEnv` low enough on the search path so that all loaded packages (except **base**) could easily access objects there.

Usage

```
temp_env()

add_items(x, y, use.names = TRUE, replace = TRUE)

add_temp(x, item, value, use.names = TRUE, replace = TRUE)

assign_temp(x, value, replace.existing = TRUE)

change_temp(x, item, value, replace.existing = TRUE)

exists_temp(x, mode = "any")

get_temp(x, default = NULL, mode = "any", item = NULL)

delete_temp(x)

rm_temp(x)

TempEnv()
```

```
addItems(x, y, use.names = TRUE, replace = TRUE)

addTemp(x, item, value, use.names = TRUE, replace = TRUE)

assignTemp(x, value, replace.existing = TRUE)

changeTemp(x, item, value, replace.existing = TRUE)

existsTemp(x, mode = "any")

getTemp(x, default = NULL, mode = "any", item = NULL)

rmTemp(x)
```

Arguments

x	The vector to add items to for add_items() or any object. for delete_temp(), it is the name of the variable (character string), or a vector of characters with the name of all variables to remove from SciViews:TempEnv.
y	The vector of which we want to inject missing items in 'x'.
use.names	Use names of items to determine which one is unique, otherwise, the selection is done on the items themselves.
replace	Do we replace existing items in 'x'?
item	The item to add data to in the list.
value	The value to add in the item, it must be a named vector and element matching is done according to name of items.
replace.existing	Do we replace an existing variable?
mode	The mode of the seeked variable
default	The default value to return, in case the variable or the item does not exist.

Details

The temporary environment is attached to the search path for easier access to its objects.

Value

The temporary environment for temp-env(), the value assigned, added or changed for assign_temp(), add_temp(), change_temp(), or get_temp(). TRUE or FALSE for exists_temp(), delete_temp() or rm_temp().

See Also

assign(), search(), temp_var()

Examples

```
ls(temp_env())

# I have a vector v1 with this:
v1 <- c(a = "some v1 text", b = "another v1 text")
# I want to add items whose name is missing in v1 from v2
v2 <- c(a = "v2 text", c = "the missign item")
add_items(v1, v2, replace = FALSE)
# Not the same as
add_items(v1, v2, replace = TRUE)
# This yield different result (names not used and lost!)
add_items(v1, v2, use.names = FALSE)

add_temp("tst", "item1", c(a = 1, b = 2))
# Retrieve this variable
get_temp("tst")
# Add to item1 in this list without replacement
add_temp("tst", "item1", c(a = 45, c = 3), replace = FALSE)
get_temp("tst")
# Same but with replacement of existing items
add_temp("tst", "item1", c(a = 45, c = 3), replace = TRUE)
get_temp("tst")
# Delete the whole variable
delete_temp("tst")

assign_temp("test", 1:10)
# Retrieve this variable
get_temp("test")

change_temp("tst", "item1", 1:10)
# Retrieve this variable
get_temp("tst")
# Create another item in the list
change_temp("tst", "item2", TRUE)
get_temp("tst")
# Change it
change_temp("tst", "item2", FALSE)
get_temp("tst")
# Delete it (= assign NULL to the item)
change_temp("tst", "item2", NULL)
get_temp("tst")
# Delete the whole variable
delete_temp("tst")

assign_temp("test", 1:10)
# Check if this variable exists
exists_temp("test")
# Remove it
delete_temp("test")
# Does it still exists?
exists_temp("test")
```

temp_var *Get an arbitrary name for a temporary variable*

Description

This function ensures that the variable name is cryptic enough and is not already used.

Usage

```
temp_var(pattern = ".var")

tempvar(pattern = ".var")
```

Arguments

pattern	The prefix for the variable (the rest is a random number).

Value

A string with the name of a variable.

See Also

tempfile()

Examples

```
temp_var()
```

to_rjson *Convert R object to and from RJSON specification*

Description

RJSON is an object specification that is not unlike JSON, but better adapted to represent R objects (i.e., richer than JSON). It is also easier to parse and evaluate in both R and JavaScript to render the objects in both languages. RJSON objects are used by SciViews to exchange data between R and SciViews GUIs like Komodo/SciViews-K.

Usage

```
to_rjson(x, attributes = FALSE)

eval_rjson(rjson)

list_to_json(x)

toRjson(x, attributes = FALSE)

evalRjson(rjson)

listToJson(x)
```

Arguments

x	Any R object to be converted into RJSON (do not work with objects containing C pointers, environments, promises or expressions, but should work with almost all other R objects).
attributes	If FALSE (by default), a simple object is created by ignoring all attributes. This is usally the suitable option to transfer data to another language, like JavaScript that do not understand R attributes anyway. With attributes = TRUE, the complete information about the object is written, so that the object could be recreated (almost) identical when evaluated in R (but prefer save() and load() to tranfer objects between R sessions!).
rjson	A string containing an object specified in RJSON notation. The specification is evaluated in R... and it can contain also R code. There is no protection provided against execution of bad code. So, you must trust the source!

Details

JSON (JavaScript Object Notation) allows to specify fairly complex objects that can be rather easily exchanged between languages. The notation is also human-readable and not too difficult to edit manually (although not advised, of course). However, JSON has too many limitations to represent R objects (no NA versus NaN, no infinite numbers, no distinction between lists and objects with attributes, or S4 objects, etc.). Moreover, JSON is not very easy to interpret in R and the existing implementations can convert only specified objects (simple objects, lists, data frames, ...).

RJSON slighly modifies and enhances JSON to make it: (1) more complete to represent almost any R object (except objects with pointers, environments, ..., of course), and (2) to make it very easy to parse and evaluate in both R and JavaScript (and probably many other) languages.

With attributes = FALSE, factors and Dates are converted to their usual character representation before encoding the RJSON object. If attributes = TRUE, they are left as numbers and their attributes (class, -and levels for factor-) completely characterize them (i.e., using eval_rjson() and such objects recreate factors or Dates, respectively). However, they are probably less easy to handle in JavaScript of other language where you import the RJSON representation.

Note also that a series of objects are not yet handled correctly. These include: complex numbers, the different date flavors other that Date, functions, expressions, environments, pointers. Do not use such items in objects that you want to convert to RJSON notation.

A last restriction: you cannot have any special characters like linefeed, tabulation, etc. in names. If you want to make your names most compatible with JavaScript, note that the dot is not allowed in syntactically valid names, but the dollar sign is allowed.

Value

For `to_rjson()`, a character string vector with the RJSON specification of the argument.

For `eval_rjson()`, the corresponding R object in case of a pure RJSON object specification, or the result of evaluating the code, if it contains R commands (for instance, a RJSONp -RJSON with padding- item where a RJSON object is an argument of an R function that is evaluated. In this case, the result of the evaluation is returned).

For `list_to_json()`, correct (standard) JSON code is generated if `x` is a list of character strings, or lists.

See Also

`parse_text()`

Examples

```
# A complex R object
obj <- structure(list(
  a = as.double(c(1:5, 6)),
  LETTERS,
  c = c(c1 = 4.5, c2 = 7.8, c3 = Inf, c4 = -Inf, NA, c6 = NaN),
  c(TRUE, FALSE, NA),
  e = factor(c("a", "b", "a")),
  f = 'this is a "string" with quote',
  g = matrix(rnorm(4), ncol = 2),
  `h&$@` = data.frame(x = 1:3, y = rnorm(3),
    fact = factor(c("b", "a", "b"))),
  i = Sys.Date(),
  j = list(1:5, y = "another item")),
  comment = "My comment",
  anAttrib = 1:10,
  anotherAttrib = list(TRUE, y = 1:4))

# Convert to simplest RJSON, without attributes
rjson1 <- to_rjson(obj)
rjson1
eval_rjson(rjson1)

# More complex RJSON, with attributes
rjson2 <- to_rjson(obj, TRUE)
rjson2
obj2 <- eval_rjson(rjson2)
obj2
# Numbers near equivalence comparison (note: identical(Robj, Robj2) is FALSE)
all.equal(obj, obj2)

rm(obj, obj2, rjson1, rjson2)
```

Index