Package 'spsUtil'

October 14, 2022

Title 'systemPipeShiny' Utility Functions

Version 0.2.2

Date 2021-10-29

Description The systemPipeShiny (SPS) framework comes with many useful utility functions. However, installing the whole framework is heavy and takes some time. If you like only a few useful utility functions from SPS, install this package is enough.

Depends R (>= 4.0.0)

Imports httr, assertthat, stringr, glue, magrittr, crayon, utils, R6, stats

Suggests testthat

License GPL (>= 3)

Encoding UTF-8

BugReports https://github.com/lz100/spsUtil/issues

URL https://github.com/lz100/spsUtil

RoxygenNote 7.1.2

Config/testthat/edition 3

NeedsCompilation no

Author Le Zhang [aut, cre]

Maintainer Le Zhang <lezhang100@gmail.com>

Repository CRAN

Date/Publication 2021-10-30 22:30:02 UTC

R topics documented:

`checkNameSpace` *check namespace*

Description

Help you to check if you have certain packages and return missing package names

Usage

```
checkNameSpace(
  packages,
  quietly = FALSE,
  from = "CRAN",
  time_out = 1,
  on_timeout = {     FALSE }
)
```

Arguments

`packages`	vector of strings
`quietly`	bool, give you warning on fail?
`from`	string, where this package is from like, "CRAN", "GitHub", only for output message display purpose
`time_out`	numeric, how long to wait before reaching the time limit. Sometimes there are too many pkgs installed and takes too long to scan the whole list. Set this timeout in seconds to prevent the long waiting.
`on_timeout`	expressions, call back experssions to run when reaches timeout time. Default is return `FALSE` as indicating that package is missing (we can't find the package).

Value

vector of strings, of missing package names, `character(0)` if no missing

Examples

```
checkNameSpace("ggplot2")
checkNameSpace("random_pkg")
checkNameSpace("random_pkg", quietly = TRUE)
```

checkUrl *check if an URL can be reached*

Description

check if a URL can be reached, return TRUE if yes and FALSE if cannot or with other status code

Usage

```
checkUrl(url, timeout = 5)
```

Arguments

url	string, the URL to request
timeout	seconds to wait before return FALSE

Value

TRUE if url is reachable, FALSE if not

Examples

```
checkUrl("https://google.com")
try(checkUrl("https://randomwebsite123.com", 1))
```

historyStack *history stack structure and methods*

Description

Some methods for a history stack data structure. It can store history of certain repeating actions. For example, building the back-end of a file/image editor, allow undo/redo actions.

Details

1. If the stack reaches the limit and you are trying to add more history, the first history step will be removed , all history will be shift to the left by one step and finally add the new step to the end.
2. When history returning methods are called, like the `get()`, `forward()`, `backward()` methods, it will not directly return the item saved, but a list, contains 4 components: 1. item, the actual item stored; 2. pos, current posistion value; 3. first, boolean value, if this history is stored on the first position of stack; 4. last, boolean value, if this history is stored on the last position of stack;
3. If you `forward` beyond last step, or `backward` to prior the first step, it will be stopped with errors.

4. Starting history stack with no initial history will return a special stack, where the `pos = 0`, `len = 0`, `first = TRUE`, and `last = TRUE`. This means you cannot move forward or backward. When you `get()`, it will be an empty list `list()`. After adding any new history, `pos` will never be 0 again, it will always be a larger than 0 value.

Value

an R6 class object

Methods

Public methods:

- `historyStack$new()`
- `historyStack$clear()`
- `historyStack$get()`
- `historyStack$getPos()`
- `historyStack$status()`
- `historyStack$forward()`
- `historyStack$backward()`
- `historyStack$add()`
- `historyStack$clone()`

Method `new()`**:** create the history object

Usage:

```
historyStack$new(items = NULL, limit = 25, verbose = TRUE)
```

Arguments:

`items` list, initial history step items to store on start

`limit` int, how many history steps can be stored in the stack, default 25 steps

`verbose` bool, print some verbose message?

Method `clear()`**:** clear all history steps in the stack

Usage:

```
historyStack$clear()
```

Method `get()`**:** retrieve the history from a certain position in the stack

Usage:

```
historyStack$get(pos = private$pos)
```

Arguments:

`pos` int, which position to get the history from, default is current step.

Method `getPos()`**:** get current step position in the history stack

Usage:

```
historyStack$getPos()
```

Method `status()`**:** print out some status of the stack

Usage:

```
historyStack$status()
```

Returns: returns a list of `pos`: current position (int); `len`: current length of the history stack (int); `limit`: history stack storing limit (int); `first`: is current step position the first of the stack (bool); `last`: is current step position the last of the stack (bool)

Method `forward()`**:** move one step forward in the history stack and return item in that position

Usage:

```
historyStack$forward()
```

Method `backward()`**:** move one step backward in the history stack and return item in that position

Usage:

```
historyStack$backward()
```

Method `add()`**:** Add an item to the history and move one step forward

Usage:

```
historyStack$add(item)
```

Arguments:

`item` any object you want to add to the stack. Everything store in the item will be moved into a list, so even if item may be something length > 1, it will still be treated as a single item and single history step.

Details: If current position is not the last position, and when a new step item is added to the stack, all history records (items) after current position will be removed before adding the new history item.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
historyStack$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
his <- historyStack$new()
# add some history
his$add(1)
his$add(2)
his$add(3)
his$add(4)
his$add(5)
# check status
his$status()
# get item at current history position
his$get()
# go back to previous step
his$backward()
```

```
# going back to step 2
his$backward()
his$backward()
# going forward 1 step tp step 3
his$forward()
# check current status
his$status()
# adding a new step at position 3 will remove the old step 4,5 before adding
his$add("new 4")
# only 3 steps + 1 new step = 4 steps left
his$status()
```

`inc` *In-place operations*

Description

In-place operations like `i += 1`, `i -= 1` is not support in R. These functions implement these operations in R.

Usage

```
inc(e1, e2 = 1)

mult(e1, e2 = 2)

divi(e1, e2 = 2)
```

Arguments

`e1`	object, most likely a numeric object
`e2`	the operation value, the value to add, subtract, multiply, divide of.

Details

`inc(i)` is the same as `i <- i + 1`. `inc(i, -1)` is the same as `i <- i - 1`. `mult(i)` is the same as `i <- i * 2`. `divi(i)` is the same as `i <- i / 2`.

Value

No return, directly assign the value back to `e1`

See Also

If you want shiny::reactiveVal version of these operators, check spsComps. shiny::reactiveValues operation will be the same as normal values.

Examples

```
i <- 0
inc(i) # add 1
i
inc(i) # add 1
i
inc(i, -1) # minus 1
i
inc(i, -1) # minus 1
i
x <- 1
mult(x) # times 2
x
mult(x) # times 2
x
divi(x) # divide 2
x
divi(x) # divide 2
x
```

`msg` *pretty logging message*

Description

If

1. `use_color` = `TRUE` or
2. under SPS main package `use_crayon`option is `TRUE`
3. In a console that supports colors

Then the message will be colorful, other wise no color.

"INFO" level spawns `message`, "WARNING" is `warning`, "ERROR" spawns `stop`, other levels use `cat`.

`spsinfo`, `spswarn`, `spserror` are higher level wrappers of `msg`. The only difference is they have `SPS-` prefix.

`spsinfo` has an additional arg `verbose`. This arg works similarly to all other `verbose` args in SPS:

1. if not specified, it follows the project option. If SPS option `verbose` is set to `TRUE`, message will be displayed; if `FALSE`, mute the message.
2. It can be be forced to `TRUE` and `FALSE`. `TRUE` will forcibly generate the msg, and `FALSE` will mute the message.

Usage

```
msg(
  msg,
  level = "INFO",
  .other_color = NULL,
  info_text = "INFO",
  warning_text = "WARNING",
  error_text = "ERROR",
  use_color = TRUE
)

spsinfo(msg, verbose = NULL)

spswarn(msg)

spserror(msg)
```

Arguments

msg	a character string of message or a vector of character strings, each item in the vector presents one line of words
level	typically, one of "INFO", "WARNING", "ERROR", not case sensitive. Other custom levels will work too.
.other_color	hex color code or named colors, when levels are not in "INFO", "WARNING", "ERROR", this value will be used
info_text	info level text prefix, use with "INFO" level
warning_text	warning level text prefix, use with "WARNING" level
error_text	error level text prefix, use with "ERROR" level
use_color	bool, default TRUE, to use color if supported?
verbose	bool, default get from sps project options, can be overwritten

Details

1. If use_color is TRUE, output message will forcibly use color if the console has color support, ignore SPS use_crayon option.
2. If use_color is FALSE, but you are using within SPS framework, the use_crayon option is set to TRUE, color will be used.
3. Otherwise message will be no color.

Value

see description and details

Examples

```
msg("this is info")
msg("this is warning", "warning")
try(msg("this is error", "error"))
msg("this is another level", "my level", "green")
spsinfo("some msg, verbose false", verbose = FALSE) # will not show up
spsinfo("some msg, verbose true", verbose = TRUE)
spswarn("sps warning")
try(spserror("sps error"))
```

`notFalsy` *Judgement of falsy value*

Description

judge if an object is or not a falsy value. This includes: empty value, empty string `""`, `NULL`, `NA`, length of 0 and `FALSE` itself

Usage

```
notFalsy(x)

isFalsy(x)

emptyIsFalse(x)
```

Arguments

`x`	any R object

Details

R does not have good built-in methods to judge falsy values and these kind of values often cause errors in `if` conditions, for example `if(NULL) 1 else 2` will cause error. So this function will be useful to handle this kind of situations: `if(notFalsy(NULL)) 1 else 2`.

1. not working on S4 class objects.
2. `isFalsy` is the reverse of `notFalsy`: `isFalsy(x) = !notFalsy(x)`
3. `emptyIsFalse` is the old name for `notFalsy`

Useful for if statement. Normal empty object in if will spawn error. Wrap the expression with `emptyIsFalse` can avoid this. See examples

Value

`NA`, `""`, `NULL`, `length(0)`, `nchar == 0` and `FALSE` will return `FALSE`, otherwise `TRUE` in `notFalsy` and the opposite in `isFalsy`

Examples

```
notFalsy(NULL)
notFalsy(NA)
notFalsy("")
try(`if(NULL) "not empty" else "empty"`) # this will generate error
if(notFalsy(NULL)) "not falsy" else "falsy" # but this will work
# Similar for `NA`, `""`, `character(0)` and more
isFalsy(NULL)
isFalsy(NA)
isFalsy("")
```

`quiet` *Suppress cat, print, message and warning*

Description

Useful if you want to suppress cat, print, message and warning. You can choose what to mute. Default all four methods are muted.

Usage

```
quiet(x, print_cat = TRUE, message = TRUE, warning = TRUE)
```

Arguments

`x`	function or expression or value assignment expression
`print_cat`	bool, mute `print` and `cat`?
`message`	bool, mute `messages`?
`warning`	bool, mute `warnings`?

Value

If your original functions has a return, it will return in `invisible(x)`

Examples

```
quiet(warning(123))
quiet(message(123))
quiet(print(123))
quiet(cat(123))
quiet(warning(123), warning = FALSE)
quiet(message(123), message = FALSE)
quiet(print(123), print_cat = FALSE)
quiet(cat(123), print_cat = FALSE)
```

remove_ANSI *Remove ANSI color code*

Description

Remove ANSI pre-/suffix-fix in a character string.

Usage

```
remove_ANSI(strings)
```

Arguments

strings	strings, a character vector

Value

strings with out ANSI characters

Examples

```
remove_ANSI("\033[34m\033[1ma\033[22m\033[39m")
remove_ANSI(c("\033[34m\033[1ma\033[22m\033[39m",
              "\033[34m\033[1mb\033[22m\033[39m"))
```

simepleStack *A simple stack structure and methods*

Description

A simple stack data structure in R, with supporting of assiocated methods, like push, pop and others.

Value

an R6 class object

Methods

Public methods:

- simepleStack$new()
- simepleStack$len()
- simepleStack$get()
- simepleStack$clear()
- simepleStack$push()
- simepleStack$pop()

- simepleStack$clone()

Method `new()`**:** initialize a new object

Usage:

```
simepleStack$new(items = list(), limit = Inf)
```

Arguments:

`items` list, list of items to add to the initial stack

`limit` int, how many items can be pushed to the stack, default is unlimited.

Method `len()`**:** returns current length of the stack

Usage:

```
simepleStack$len()
```

Method `get()`**:** returns the full current stack of **all items**

Usage:

```
simepleStack$get()
```

Method `clear()`**:** remove **all items** in current stack

Usage:

```
simepleStack$clear()
```

Method `push()`**:** add item(s) to the stack

Usage:

```
simepleStack$push(items, after = self$len())
```

Arguments:

`items` list, list of items to add to the stack

`after` int, which position to push items after, default is after the current last item. 0 will be before the first item.

Method `pop()`**:** remove item(s) from the stack and return as results

Usage:

```
simepleStack$pop(len = 1, tail = FALSE)
```

Arguments:

`len` int, how many items to pop from stack, default is 1 item a time.

`tail` bool, to pop in the reverse order (from the last item)? Default is `FALSE`, pop from the top (first item).

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
simepleStack$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
my_stack <- simepleStack$new()
# check length
my_stack$len()
# add some thing
my_stack$push(list(1, 2, 3))
# print current stack
str(my_stack$get())
# check length
my_stack$len()
# add before the current first
my_stack$push(list(0), after = 0)
# print current stack
str(my_stack$get())
# pop one item
my_stack$pop()
# print current stack
str(my_stack$get())
# pop one item from the tail
my_stack$pop(tail = TRUE)
# print current stack
str(my_stack$get())
# pop more than one items
my_stack$pop(2)
# print current stack
str(my_stack$get()) # nothing left
```

`spsOption` *Get or set SPS options*

Description

Some functions in spsUtil, spsComps and systempPipeShiny will behave differently if some SPS options are changed, but it is optional. All functions have a default value. If SPS options are not changed, they will just use the default setting. Read help files of individual functions for detail.

Usage

```
spsOption(opt, value = NULL, .list = NULL, empty_is_false = TRUE)
```

Arguments

`opt`	string, length 1, what option you want to get or set
`value`	if this is not `NULL`, this function will set the option you choose to this value
`.list`	list, set many SPS options together at once by passing a list to this function.
`empty_is_false`	bool, when trying to get an option value, if the option is `NULL`, `NA`, `""` or length is 0, return `FALSE`?

Value

return the option value if value exists; return FALSE if the value is empty, like NULL, NA, ""; return NULL if empty_is_false = FALSE; see notFalsy

If value != NULL will set the option to this new value, no returns.

Examples

```
spsOption("test1") # get a not existing option
spsOption("test1", 1) # set the value
spsOption("test1") # get the value again
spsOption("test2")
spsOption("test2", empty_is_false = FALSE)
spsOption(.list = list(
    test1 = 123,
    test2 = 456
))
spsOption("test1")
spsOption("test2")
```

---

strUniquefy *Uniquefy a character vector*

Description

Fix duplicated values in a character vector, useful in column names and some ID structures that requires unique identifiers. If any duplicated string is found in the vector, a numeric index will be added after the these strings.

Usage

```
strUniquefy(x, sep_b = "_", sep_a = "", mark_first = TRUE)
```

Arguments

x	character vector
sep_b	string, separator before the number index
sep_a	string, separator after the number index
mark_first	bool, if duplicated values are found, do you want to add the numeric index starting from the first copy? FALSE means starting from the second copy.

Details

The input can also be a numeric vector, but the return will always be character.

Value

returns a character vector

Examples

```
strUniquefy(c(1,1,1,2,3))
strUniquefy(c(1,1,1,2,3), mark_first = FALSE)
strUniquefy(c(1,1,1,2,3), sep_b = "(", sep_a = ")")
strUniquefy(c("a","b","c","a","d","b"))
```

---

timeout *Run expressions with a timeout limit*

Description

Add a time limit for R expressions

Usage

```
timeout(
  expr,
  time_out = 1,
  on_timeout = {     stop("Timout reached", call. = FALSE) },
  on_final = { },
  env = parent.frame()
)
```

Arguments

expr	expressions, wrap them inside {}
time_out	numeric, timeout time, in seconds
on_timeout	expressions, callback expressions to run it the time out limit is reached but expression is still running. Default is to return an error.
on_final	expressions, callback expressions to run in the end regardless the state and results
env	environment, which environment to evaluate the expressions. Default is the same environment as where the timeout function is called.

Details

Expressions will be evaluated in the parent environment by default, for example if this function is called at global level, all returns, assignments inside expr will directly go to global environment as well.

Value

default return, all depends on what return the expr will have

Examples

```
# The `try` command in following examples are here to make sure the
# R CMD check will pass on package check. In a real case, you do not
# need it.

# default
try(timeout({Sys.sleep(0.1)}, time_out = 0.01))
# timeout is evaluating expressions the same level as you call it
timeout({abc <- 123})
# so you should get `abc` even outside the function call
abc
# custom timeout callback
timeout({Sys.sleep(0.1)}, time_out = 0.01, on_timeout = {print("It takes too long")})
# final call back
try(timeout({Sys.sleep(0.1)}, time_out = 0.01, on_final = {print("some final words")})) # on error
timeout({123}, on_final = {print("runs even success")})  # on success
# assign to value
my_val <- timeout({10 + 1})
my_val
```

Index