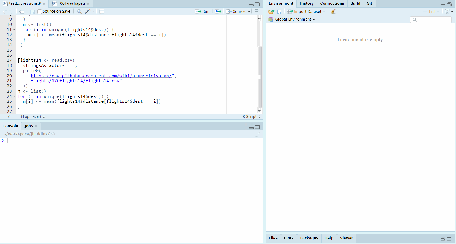
**Introduction**

Profiling our code is a very useful tool to determine how well the code performs on different metrics.

**The addin we will create in this article will let us use a keyboard shortcut to run profiling on R code selected in RStudio without blocking the session or requiring any external packages.**

Specifically for very simple overview use, it may be beneficial to look at the time needed for a set of expressions to compute, e.g. how fast the code is. Secondly, especially important in case of computing on big datasets in-memory, the amount of memory utilized, e.g. how much RAM was used.



The addin in action

**Profiling options provided by base R**

Without going into any detail at all, we have 2 very nice options to profile our code with base R:

* base::system.time(expr) – returns CPU and other times that expr used
* utils::Rprof – can serve as a switch to enable and disable profiling, with a variety of options, saving the results into a file on disk, by default "Rprof.out"

For the use of our addin, we will utilize the second approach, as we are interested not only in time spent, but also in memory utilization of the profiled expressions.

After finishing the profiling, we will use utils::summaryRprof to summarize the results provided to us by the Rprof functionality mentioned above. To get an overview, we will examine only the total time the selected expressions took to execute and the maximum memory.

The very simplistic implementation can look as follows:

profileExpression <- function(expr) {

on.exit({

unlink("Rprof.out")

utils::Rprof(NULL)

})

if (!is.expression(expr)) {

message("epxr must be an expression in profileExpression()")

return(data.frame(

totalTime = numeric(0),

maxMemory = numeric(0)

))

}

gc()

utils::Rprof(

memory.profiling = TRUE,

interval = 0.01,

append = FALSE

)

evalRes <- try(eval(expr), silent = TRUE)

utils::Rprof(NULL)

if (inherits(evalRes, "try-error")) {

return(data.frame(stringsAsFactors = FALSE,

totalTime = "EvalError",

maxMemory = "EvalError"

))

}

res <- utils::summaryRprof(memory = "both")

data.frame(

totalTime = max(res[["by.total"]][, 1L]),

maxMemory = max(res[["by.total"]][, 5L])

)

}

Since we maybe be interested in more than one execution of the expressions to be profiled and the profiling will be running in background, a wrapper executing the profiling itself multiple times may come in handy. Except the number of times to execute, which is a very standard argument, we can also attempt to provide a time frame we want to invest into the profiling:

multiProfile <- function(

expr,

times = 10L,

maxtime = getOption("jhaddins\_profiler\_maxtime", default = NULL)

){

if (!(is.integer(times) || is.integer(maxtime))) {

message("Times or maxtime must be integer in multiProfile()")

return(data.frame(

totalTime = numeric(0),

maxMemory = numeric(0)

))

}

first <- profileExpression(expr)

if (!is.null(maxtime)) {

if (is.numeric(first[["totalTime"]])) {

times <- floor(maxtime / first[["totalTime"]])

} else {

message("Eval failed, cannot compute times from maxtime.")

return(first)

}

}

if (times <= 1L) {

return(first)

}

rest <- do.call(

rbind,

lapply(rep(list(expr), times - 1L), profileExpression)

)

rbind(first, rest)

}

**Asynchronous execution and communication of the results with the session**

Since we are only using base R functionality without taking advantage of external packages that would help us execute the profiling asynchronously, we have 3 challenges:

1. Asynchronous execution of the profiling

We can take advantage of base R’s convenient interface system2, which allows us to invoke OS commands, with the option to run asynchronously providing wait = FALSE as argument.

1. Communicating the results between our R session and the one running via system2

To kill two birds with one stone, we can simply use the rstudioapi to navigate to a created file, into which we will later write the profiling results using the asynchronously running process. This way we have the results immediately available within in RStudio and we can keep working conveniently on the tasks at hand. Since our application is very simple, we also avoid complications with communication between the processes for example via sockets.

1. Contents of the workspace

When selecting a code chunk to profile in RStudio, it will likely happen very soon that the execution of expressions included in the selected code will rely on the current state of the global environment (aka. workspace). We can therefore make our functionality more convenient by storing the contents of the global environment on disk and loading it before running the profiler in our asynchronous process.

A simple example implementation of the thoughts above it once again presented below. Note that this implementation is very bare-bones and could use much polishing, which may happen sometime after publishing this article:

runProfiler <- function(

inpContext = rstudioapi::getActiveDocumentContext()

){

force(inpContext)

inpString <- inpContext[["selection"]][[1L]][["text"]]

cat(inpString, file = file.path("~/temp.R"))

expr <- try(parse("~/temp.R"), silent = TRUE)

if (inherits(expr, "try-error")) {

message("Selected text cannot be parsed, cannot profile.")

unlink(file.path("~/temp.R"))

return(1L)

}

save(

list = ls(all.names = TRUE, envir = .GlobalEnv),

file = "~/tmp.RData",

envir = .GlobalEnv

)

script <- paste(sep = "; ",

"load('~/tmp.RData')",

"res <- jhaddins:::multiProfile(parse('~/temp.R'))",

"jhaddins:::writeProfileDf(res)",

"unlink('~/temp.R')",

"unlink('~/tmp.RData')"

)

file.create("~/tmp\_prof.txt")

rstudioapi::navigateToFile("~/tmp\_prof.txt")

system2(

command = 'Rscript',

args = c('-e', shQuote(script)),

wait = FALSE

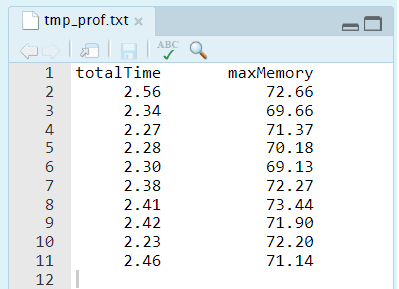
)

message("Profiler running in the background")

}

**Results of the profiling**

For the use that this simple functionality was developed, the main interest is knowing 2 very simple sets of information – how fast did the expressions execute and how much maximum memory was utilized. This is why the results are extracted and written in an extremely simplistic way, as can be seen below:



“quand il n’y a plus rien à retrancher”

Based on real-life usage we may still improve the presentation (a bit in the future.

**The addin formalities**

If you follow this blog for a bit, you can safely skip this part. A few things to make our new addin available and easy to use:

1. Add the addin bindings into inst/addins.dcf

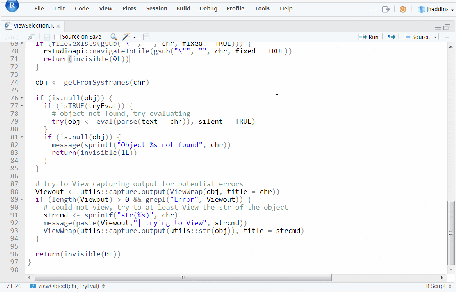
Name: runProfiler

Description: experimental, runProfiler

Binding: runProfiler

Interactive: false

1. Re-install the package
2. Assign a keyboard shortcut in the Tools -> Addins -> Browse Addins... -> Keyboard Shortcuts... menu in RStudio:



Assigning a keyboard shortcut to use the Addin

Addroxytag.R

#' roxyfy

#'

#' @description Encloses \code{str} into \code{\\tag{}}.

#'

#' @param str \code{character(1)}, string to enclose.

#' @param tag \code{character(1)}, to enclose \code{str} with.

#' @param splitLines \code{logical(1)}, if \code{TRUE},

#' splits \code{str} by \code{\\n} and applies the tag

#' to each element.

#'

#' @return \code{character(1)}, the input \code{str} enclosed

#' into the provided \code{tag}.

#'

#' @examples \dontrun{roxyfy("character", "code")}

roxyfy <- function(str, tag = NULL, splitLines = TRUE) {

if (is.null(tag)) {

return(str)

}

if (!isTRUE(splitLines)) {

return(paste0("\\", tag, "{", str, "}"))

}

str <- unlist(strsplit(str, "\n"))

str <- paste0("\\", tag, "{", str, "}")

paste(str, collapse = "\n")

}

#' addRoxytag

#'

#' @description Intended to be used as an RStudio addin. When

#' executed, adds a tag around the selections in the active

#' RStudio document using \code{\link{roxyfy}}. See the

#' wrappers \code{\link{addRoxytagCode}},

#' \code{\link{addRoxytagLink}}, \code{\link{addRoxytagEqn}}

#' for examples.

#'

#' @param tag \code{character(1)}, roxygen tag to use.

#'

#' @importFrom rstudioapi getActiveDocumentContext modifyRange

#'

#' @seealso \code{\link{roxyfy}}, \code{\link{addRoxytagCode}},

#' \code{\link{addRoxytagLink}}, \code{\link{addRoxytagEqn}}

#'

#' @return Side-effects caused by addin execution.

addRoxytag <- function(tag = NULL) { # nocov start

context <- rstudioapi::getActiveDocumentContext()

lapply(X = context[["selection"]]

, FUN = function(thisSel, contextid) {

rstudioapi::modifyRange(location = thisSel[["range"]]

, roxyfy(thisSel[["text"]], tag)

, id = contextid)

}

, contextid = context[["id"]]

)

return(invisible(NULL))

} # nocov end

#' addRoxytagCode

#'

#' @description Intended to be used as an RStudio addin. When

#' executed, adds a tag \code{"code{}"} around the selections

#' in the active RStudio document

#'

#' @seealso \code{\link{addRoxytag}}

addRoxytagCode <- function() { # nocov start

addRoxytag(tag = "code")

} # nocov end

#' addRoxytagLink

#'

#' @description Intended to be used as an RStudio addin. When

#' executed, adds a tag \code{"link{}"} around the selections

#' in the active RStudio document

#'

#' @seealso \code{\link{addRoxytag}}

addRoxytagLink <- function() { # nocov start

addRoxytag(tag = "link")

} # nocov end

#' addRoxytagEqn

#'

#' @description Intended to be used as an RStudio addin. When

#' executed, adds a tag \code{"eqn{}"} around the selections

#' in the active RStudio document

#'

#' @seealso \code{\link{addRoxytag}}

addRoxytagEqn <- function() { # nocov start

addRoxytag(tag = "eqn")

} # nocov end

Exportdataframe.R

exportDataFrame <- function() { # nocov start

context <- rstudioapi::getActiveDocumentContext()

dt <- getFromSysframes(context[["selection"]][[1]][["text"]])

tmpF <- tempfile()

write.table(dt, file = tmpF, sep = "\t")

rstudioapi::navigateToFile(tmpF)

x <- rstudioapi::getSourceEditorContext()

while (!(grepl(tmpF, x$path, fixed = TRUE))) {

Sys.sleep(0.2)

x <- rstudioapi::getSourceEditorContext()

}

rng <- structure(

list(

start = structure(c(row = 1, column = 1), class = "document\_position"),

end = structure(c(row = Inf, column = Inf), class = "document\_position")

),

class = "document\_range"

)

rstudioapi::setSelectionRanges(rng, id = x$id)

} # nocov end

InsertPostToc.R

#' Create a table of contents for a blogdown post

#'

#' @description Creates a markdown list from section level divs and

#' <h> tags in a html file that was generated by the blogdown package.

#'

#' @param path \code{character(1)}, path to an html file to be read.

#' @param ordered \code{logical(1)}, if \code{TRUE} an ordered list

#' is created resulting in \code{<ol>}), non-ordered otherwise,

#' resulting in \code{<ul>}.

#'

#' @return \code{character()}, a markdown-based table of contents,

#' one element per heading.

makePostToc <- function(path, ordered = TRUE) {

content <- readLines(

path,

encoding = getOption("jhaddins\_posttoc\_encoding", default = "UTF-8")

)

# get the html links from <div ... class="section level">

links <- grep('class="section level', content, fixed = TRUE, value = TRUE)

links <- gsub('<div id="', '', links, fixed = TRUE)

links <- gsub('\" class=\"section level[1-9]{1}\">', '', links)

# get the heading labels for <h> tags

labls <- grep('<h[1-9]{1}>', content, value = TRUE)

# intendation based on heading level

levels <- as.integer(

gsub('[^0-9.]', '', regmatches(labls, regexpr('<h[1-9]{1}>', labls)))

)

prefixes <- strrep(" ", 4L \* (levels - 1L))

# strip the heading labels of html tags

labls <- gsub('</?h[1-9]{1}>', '', labls)

labls <- gsub('</?[a-z]\*>', '', labls)

# list type (ordered or not)

lt <- if (isTRUE(ordered)) '1.' else '\*'

paste0(prefixes, lt, ' [', labls, '](#', links, ')')

}

#' Check if context is feasible to generate a table of contents

#'

#' @description Check if an RStudio document context is feasible

#' for table of contents generation with the \code{\link{insertPostToc}}

#' addin.

#' @param ctx \code{list}, context of document selection objects,

#' usually provided by \code{\link{getActiveDocumentContext}}.

#'

#' @return \code{logical(1)}, \code{TRUE} if table of contents can be

#' generated, \code{FALSE} otherwise.

checkTocCtx <- function(ctx) {

fn <- ctx[["path"]]

if (!tools::file\_ext(fn) %in% c('Rmd', 'md')) {

message('Use with a .Rmd or .md file open and active in RStudio editor')

return(FALSE)

}

ht <- sub(tools::file\_ext(fn), 'html', fn, fixed = TRUE)

if (!file.exists(ht)) {

message(paste(

'File named: ', ht, 'does not exist, cannot read the links from it.',

'Did you run blogdown::serve\_site() or build\_site() to generate it?'

))

return(FALSE)

}

TRUE

}

#' Create a list of arguments to be passed to rstudioapi::insertText

#'

#' @description Based on the provided RStudio's context, create a

#' list of arguments that can be passed to \code{rstudioapi::insertText}

#' via \code{do.call}.

#'

#' @details This is done for better testability. The addin function itself

#' \code{\link{insertPostToc}} wraps around this function.

#'

#' @param ctx \code{list}, context of document selection objects,

#' usually provided by \code{\link{getActiveDocumentContext}}.

#'

#' @return \code{list} of 2 - \code{location} specifies the starting

#' position at which the table of contents should be inserted and

#' \code{text} holds the content to be inserted.

getTocArgs <- function(ctx) {

fn <- ctx[["path"]]

ht <- sub(paste0('\\.', tools::file\_ext(fn), '$'), '.html', fn)

toc <- paste(makePostToc(ht), collapse = '\n')

loc <- ctx[["selection"]][[1L]][["range"]][["start"]]

list(location = loc, text = toc)

}

#' Insert table of contents

#'

#' @description Intended to be used as an RStudio addin. If an (R) markdown

#' file is open and active in the RStudio editor, attempts to retrieve

#' the table of contents from <h> tags in an equally-named html file

#' (generated by \code{rmarkdown::render()},

#' \code{blogdown::serve\_site()}, etc.) and insert it at cursor

#' position.

#'

#' @details The html file must be rendered first before this addin can

#' be used.

#'

#' This is useful for example in a blogdown post context, where we want to

#' easily create a table of contents that is custom and placed in an

#' arbitrary position, but copying out the links is too much hassle.

#'

#' @param ctx \code{list}, context of document selection objects, usually

#' provided by \code{\link{getActiveDocumentContext}}.

#'

#' @return Side-effects caused by addin execution.

insertPostToc <- function(ctx = rstudioapi::getSourceEditorContext()) {

if (checkTocCtx(ctx)) do.call(rstudioapi::insertText, getTocArgs(ctx))

}

MakeHighChart.R – Creating High Chart

makeHighChart <- function(

hc,

exportpath = NULL,

adddiv = TRUE,

wrapscript = TRUE,

chartname = "container",

docat = FALSE

){

if (!requireNamespace("jsonlite", quietly = TRUE)) {

message("Package jsonlite needed.")

return()

}

# Produce the js from the object --------------------------------------------

# Replicating highcharter::export\_hc with base and jsonlite

cChart <- jsonlite::toJSON(

hc$x$hc\_opts,

pretty = TRUE, auto\_unbox = TRUE, force = TRUE, null = "null"

)

cChart <- unlist(strsplit(cChart, "\n")[1])

cChart <- gsub('"NA"', 'null', cChart, fixed = TRUE)

cChart <- gsub("'NA'", 'null', cChart, fixed = TRUE)

cChart <- sub("\"", "", cChart)

cChart <- sub("\":", ":", cChart)

cChart <- paste(unlist(cChart)[-1], " ", collapse = "\n")

cChart <- gsub("\n\\s{4,}\\]\\,\n\\s{4,}\\[\n\\s{4,}", "],[", cChart)

cChart <- paste0(

"$(function () {\n $('#container').highcharts({\n",

cChart,

"\n );\n});"

)

# Wrap into (optional) niceties ---------------------------------------------

cChart <- gsub(

"$('#container')",

sprintf("$('#%s')", chartname),

cChart,

fixed = TRUE

)

if (isTRUE(wrapscript)) {

cChart <- c('<script type="text/javascript">', cChart, '</script>')

}

if (isTRUE(adddiv)) {

divtext <- sprintf('<div id="%s"></div>', chartname)

cChart <- c(cChart, "", divtext)

}

# If desired, produce optional side effects ---------------------------------

if (!is.null(exportpath)) {

writeLines(cChart, exportpath)

}

if (isTRUE(docat)) {

cat(paste(cChart, collapse = "\n"))

}

invisible(cChart)

}

Profiler.R

#' Profile an R expression

#'

#' @description Run the profiler to profile \code{expr}.

#'

#' @param expr \code{expression} to profile.

#'

#' @importFrom utils Rprof summaryRprof

#'

#' @examples \dontrun{

#' profileExpression(expression(x <- runif(100000000)))

#' }

#'

#' @return \code{data.frame} with 2 columns: \code{totalTime} in

#' seconds and \code{maxMemory} in MB.

profileExpression <- function(expr) {

tmpF <- tempfile()

on.exit({

utils::Rprof(NULL)

unlink(tmpF, force = TRUE)

})

if (!is.expression(expr)) {

message("epxr must be an expression in profileExpression()")

return(data.frame(

totalTime = numeric(0),

maxMemory = numeric(0)

))

}

gc()

utils::Rprof(

filename = tmpF,

memory.profiling = TRUE,

interval = 0.01,

append = FALSE

)

evalRes <- try(eval(expr), silent = TRUE)

utils::Rprof(NULL)

if (inherits(evalRes, "try-error")) {

return(data.frame(stringsAsFactors = FALSE,

totalTime = "EvalError",

maxMemory = "EvalError"

))

}

res <- utils::summaryRprof(filename = tmpF, memory = "both")

data.frame(

totalTime = max(res[["by.total"]][, 1L]),

maxMemory = max(res[["by.total"]][["mem.total"]])

)

}

#' Profile an R expression multiple times

#'

#' @description A wrapper around \code{\link{profileExpression}} to

#' profile an R expression multiple times or use a given time to

#' profile as many times as possible.

#'

#' @param expr \code{expression}, to be profiled.

#' @param times \code{integer(1)}, how many times to run the profiler.

#' @param maxtime \code{NULL or integer(1)}, time in seconds. If not

#' \code{NULL}, the profiler will try to estimate how many times it

#' can profile in \code{maxtime} and profile that many times,

#' effectively attempting to profile as many times as possible in

#' \code{maxtime} seconds, however at least 1 time always.

#'

#' @examples \dontrun{

#' multiProfile(expression(x <- runif(100000000)), maxtime = 60L)

#' }

#'

#' @seealso \code{\link{profileExpression}}

#'

#' @return \code{data.frame} with 2 columns: \code{totalTime} in

#' seconds and \code{maxMemory} in MB.

multiProfile <- function(

expr,

times = 10L,

maxtime = getOption("jhaddins\_profiler\_maxtime", default = NULL)

){

if (!(is.integer(times) || is.integer(maxtime))) {

message("Either times or maxtime must be integer in multiProfile()")

return(data.frame(

totalTime = numeric(0),

maxMemory = numeric(0)

))

}

first <- profileExpression(expr)

if (!is.null(maxtime)) {

if (is.numeric(first[["totalTime"]])) {

times <- floor(maxtime / first[["totalTime"]])

} else {

message("Eval failed, cannot compute times from maxtime.")

return(first)

}

}

if (times <= 1L) {

return(first)

}

rest <- do.call(

rbind,

lapply(rep(list(expr), times - 1L), profileExpression)

)

rbind(first, rest)

}

#' Write profiling results to a text file

#'

#' @param df \code{data.frame} to write.

#' @param filePath \code{character(1)} file name to write

#' \code{df} into.

#'

#' @importFrom utils write.table

#'

#' @seealso

#' \code{\link{profileExpression}}

#' \code{\link{multiProfile}}

#'

#' @return Side-effects.

writeProfileDf <- function(df, filePath = tempfile()) {

df <- as.data.frame(

stringsAsFactors = FALSE,

lapply(

df,

format,

width = 9L, digits = 2L, nsmall = 2L, scientific = FALSE

)

)

write.table(

x = df,

file = filePath,

quote = FALSE,

append = FALSE,

sep = "\t",

row.names = FALSE

)

}

#' Profile code chunk selected in RStudio

#'

#' @description Intended to be used as an RStudio addin.

#'

#' @param inpContext \code{document\_context} class object.

#' @param autoOpen \code{logical(1)} if \code{TRUE} the result

#' file will be opened in RStudio.

#' @param wait \code{logical(1)} if \code{TRUE} the main R process

#' will wait for the profiler to finish.

#'

#' @seealso

#' \code{\link{profileExpression}}

#' \code{\link{multiProfile}}

#' \code{\link{writeProfileDf}}

#'

#' @return Side-effects caused by addin execution.

runProfiler <- function(

inpContext = rstudioapi::getActiveDocumentContext(),

autoOpen = TRUE,

wait = FALSE

){

tmpContext <- tempfile(fileext = ".rds")

tmpRdata <- tempfile(fileext = ".RData")

tmpR <- tempfile(fileext = ".R")

tmpProf <- tempfile(fileext = ".txt")

force(inpContext)

saveRDS(inpContext, tmpContext)

inpString <- inpContext[["selection"]][[1L]][["text"]]

cat(inpString, file = tmpR)

expr <- try(parse(file = tmpR), silent = TRUE)

if (inherits(expr, "try-error")) {

message("Selected text cannot be parsed, cannot profile.")

unlink(tmpR, force = TRUE)

return(1L)

}

save(

list = ls(all.names = TRUE, envir = .GlobalEnv),

file = tmpRdata,

envir = .GlobalEnv

)

script <- paste(sep = "; ",

paste0("try({ load('", tmpRdata, "')"),

paste0("res <- jhaddins:::multiProfile(parse('", tmpR, "'))"),

"jhaddins:::writeProfileDf(res) })",

paste0("unlink('", tmpR, "', force = TRUE)"),

paste0("unlink('", tmpRdata, "', force = TRUE)"),

paste0("unlink('", tmpContext, "', force = TRUE)")

)

file.create(tmpProf)

if (isTRUE(autoOpen)) {

rstudioapi::navigateToFile(tmpProf)

}

if (isTRUE(wait)) {

message("Profiler starting")

} else {

message("Profiler starting in the background")

}

system2(

command = 'Rscript',

args = c('-e', shQuote(script)),

wait = wait

)

return(0L)

}

RunCurrentScript.R

runCurrentRscript <- function(

path = path.expand(rstudioapi::getActiveDocumentContext()[["path"]]),

command = "Rscript",

outputFile = tempfile(),

autoOpen = TRUE,

addRhome = TRUE

) {

if (isTRUE(addRhome)) {

command <- file.path(R.home("bin"), command)

}

system2(

command,

args = c(path, "--vanilla"),

stdout = outputFile,

stderr = outputFile

)

if (!is.null(outputFile) && file.exists(outputFile) && isTRUE(autoOpen)) {

rstudioapi::navigateToFile(outputFile)

}

}

ViewSelection.R

#' View object selected in RStudio

#'

#' @description Intended to be used as an RStudio addin.

#' Tries to view the selections in RStudio active document

#' in a meaningful way using \code{\link{getViewArgs}}.

#'

#' @param ctx \code{list}, context of document selection objects,

#' usually provided by \code{\link{getActiveDocumentContext}}.

#'

#' @importFrom rstudioapi getActiveDocumentContext navigateToFile

#' @importFrom utils capture.output str

#'

#' @seealso \code{\link{getViewArgs}}

#'

#' @return \code{list of NULL}, invisibly and side-effects caused by

#' addin execution.

viewSelection <- function(

ctx = rstudioapi::getActiveDocumentContext()

) { # nocov start

invisible(lapply(

X = ctx[["selection"]],

FUN = function(thisSel) {

callArgs <- getViewArgs(thisSel[["text"]])

do.call(what = callArgs[["what"]], args = callArgs[["args"]])

}

))

} # nocov end

#' Get object value from sys frames

#'

#' @description Looks for \code{x} in all \code{sys.frames} (except

#' the current one) and returns the value if found, \code{NULL}

#' otherwise.

#'

#' @param x \code{character(1)} string, name of the object to look for.

#'

#' @seealso

#' \code{\link{sys.frames}}

#' \code{\link{sys.nframe}}

#'

#' @examples \dontrun{

#' getFromSysframes("reshape")

#' }

#'

#' @return \code{NULL} if object with name \code{x} not found, value of

#' the object with name \code{x} if found, invisibly.

getFromSysframes <- function(x) {

if (!(is.character(x) && length(x) == 1 && nchar(x) > 0)) {

warning("Expecting a non-empty character of length 1. Returning NULL.")

return(invisible(NULL))

}

validframes <- c(rev(sys.frames()[-sys.nframe()]), .GlobalEnv)

res <- NULL

for (i in validframes) {

inherits <- identical(i, .GlobalEnv)

res <- get0(x, i, inherits = inherits)

if (!is.null(res)) {

return(res)

}

}

return(invisible(res))

}

#' Get view arguments

#'

#' @description Retrieves a function and its arguments to be used to

#' view the object defined by \code{chr} with

#' \code{\link{viewSelection}}.

#'

#' @param chr \code{character(1)} string to process.

#' @param tryEval \code{logical(1)} if \code{TRUE}, tries retrieving

#' the value of \code{chr} by parsing it and using \code{eval}.

#'

#' @examples \dontrun{

#' x <- datasets::anscombe; y <- getViewArgs("x")

#' print(getViewArgs("reshape"))

#' }

#'

#' @seealso

#' \code{\link{getFromSysframes}}

#' \code{\link{viewSelection}}

#'

#' @return \code{list} with elements \code{what} and \code{args} to use for

#' \code{do.call} in \code{\link{viewSelection}}

getViewArgs <- function(

chr,

tryEval = getOption("jhaddins\_view\_tryeval", default = TRUE)

) {

blankCall <- invisible(

list(what = function(...) invisible(NULL), args = list())

)

if (!(is.character(chr) && length(chr) == 1 && nchar(chr) > 0)) {

message("Invalid input, expecting a non-empty character of length 1")

return(blankCall)

}

# maybe it is an unquoted filename - if so, open it -------------------

if (file.exists(chr)) {

return(invisible(

list(what = rstudioapi::navigateToFile

, args = list(chr))

))

}

# or maybe it is a quoted filename - if so, open it -------------------

if (file.exists(gsub("\"", "", chr, fixed = TRUE))) {

return(invisible(

list(what = rstudioapi::navigateToFile

, args = list(gsub("\"", "", chr, fixed = TRUE))

)

))

}

# try to get the objec, see if found, try simplest eval ---------------

obj <- getFromSysframes(chr)

if (is.null(obj)) {

if (isTRUE(tryEval)) {

try(obj <- eval(parse(text = chr)), silent = TRUE)

}

if (is.null(obj)) {

message(sprintf("Object %s not found", chr))

return(blankCall)

}

}

# guess if we can use the viewer --------------------------------------

candf <- !inherits(try(as.data.frame(obj), silent = TRUE), "try-error")

if (is.list(obj) || candf || is.function(obj)) {

return(invisible(

list(what = "View"

, args = list(x = obj, title = chr))

))

} else {

# if not, view the structure ------------------------------------------

return(invisible(

list(what = "View"

, args = list(x = capture.output(str(obj)), title = chr))

))

}

}