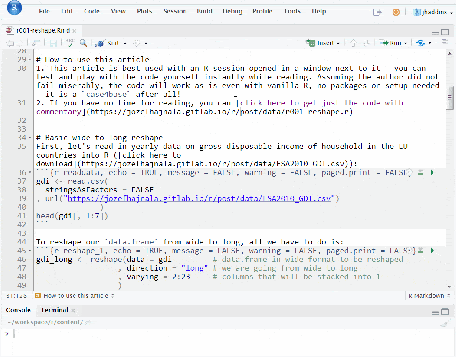
**Introduction**

In this post in the RStudio:addins series we will try to make our work more efficient with an addin for better inspection of objects, functions and files within RStudio. RStudio already has a very useful View function and a Go To Function / File feature with F2 as the default keyboard shortcut and yes, I know I promised automatic generation of @importFrom roxygen tags, unfortunately we will have to wait a bit longer for that one but I believe this one more than makes up for it in usefulness.

**The addin we will create in this article will let us use RStudio to View and inspect a wide range of objects, functions and files with 1 keypress.**

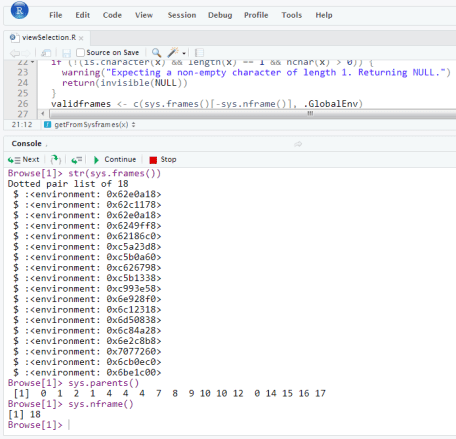


The addins in action

**Retrieving objects from sys.frames**

As a first step, we need to be able to retrieve the value of the object we are looking for based on a character string from a frame within the currently present sys.frames() for our session. This may get tricky, as it is not sufficient to only look at parent frames, because we may easily have multiple sets of “parallel” call stacks, especially when executing addins.

An example can be seen in the following screenshot, where we have a browser() call executed during the Addin execution itself. We can see that our current frame is 18 and browsing through its parent would get us to frames 17 -> 16 -> 15 -> 14 -> 0 (0 being the .GlobalEnv). The object we are looking for is however most likely in one of the other frames (9 in this particular case):



Example of sys.frames

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(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))

}

**Viewing files, objects, functions and more efficiently**

As a second step, we write a function to actually view our object in RStudio. We have quite some flexibility here, so as a first shot we can do the following:

1. Open a file if the selection (or the selection with quotes added) is a path to an existing file. This is useful for viewing our scripts, data files, etc. even if they are not quoted, such as the links in your Rmd files
2. Attempt to retrieve the object by the name and if found, try to use View to view it
3. If we did not find the object, we can optionally still try to retrieve the value by evaluating the provided character string. This carries some pitfalls, but is very useful for example for
   * viewing elements of lists, vectors, etc. where we need to evaluate [, [[ or $ to do so.
   * viewing operation results directly in the viewer, as opposed to writing them out into the console, useful for example for wide matrices that (subjectively) look better in the RStudio viewer, compared to the console output
4. If the View fails, we can still show useful information by trying to View its structure, enabling us to inspect objects that cannot be coerced to a data.frame and therefore would fail to be viewed.

viewObject <- function(chr,

tryEval = getOption("jhaddins\_view\_tryeval",

default = TRUE)

) {

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

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

return(invisible(1L))

}

ViewWrap <- get("View", envir = as.environment("package:utils"))

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

if (file.exists(chr)) {

rstudioapi::navigateToFile(chr)

return(invisible(0L))

}

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

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

rstudioapi::navigateToFile(gsub("\"", "", chr, fixed = TRUE))

return(invisible(0L))

}

obj <- getFromSysframes(chr)

if (is.null(obj)) {

if (isTRUE(tryEval)) {

# object not found, try evaluating

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

}

if (is.null(obj)) {

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

return(invisible(1L))

}

}

# try to View capturing output for potential errors

Viewout <- utils::capture.output(ViewWrap(obj, title = chr))

if (length(Viewout) > 0 && grepl("Error", Viewout)) {

# could not view, try to at least View the str of the object

strcmd <- sprintf("str(%s)", chr)

message(paste(Viewout,"| trying to View", strcmd))

ViewWrap(utils::capture.output(utils::str(obj)), title = strcmd)

}

return(invisible(0L))

}

This function can of course be improved and updated in many ways, for example using the summary method instead of str for selected object classes, or showing contents of .csv (or other data) files already read into a data.frame.

**The addin function, updating the .dcf file and key binding**

If you followed the previous posts in the series, you most likely already know what is coming up next. First, we need a function serving as a binding for the addin that will execute out viewObject function on the active document’s selections:

viewSelection <- function() {

context <- rstudioapi::getActiveDocumentContext()

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

, FUN = function(thisSel) {

viewObject(thisSel[["text"]])

}

)

return(invisible(NULL))

}

Secondly, we update the inst/rstudio/addins.dcf file by adding the binding for the newly created addin:

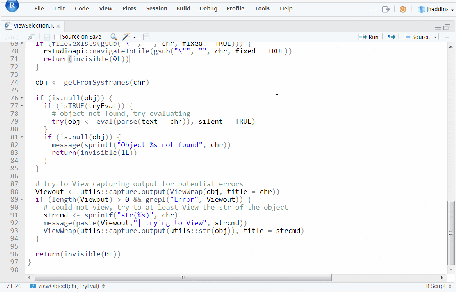
Name: viewSelection

Description: Tries to use View to View the object defined by a text selected in RStudio

Binding: viewSelection

Interactive: false

Finally, we re-install the package and assign the keyboard shortcut in the Tools -> Addins -> Browse Addins... -> Keyboard Shortcuts... menu. Personally I assigned a single F4 keystroke for this, as I use it very often:



Assigning a keyboard shortcut to use the Addin

**The addin in action**

Now, let’s view a few files, a data.frame, a function and a try-error class object just pressing F4.

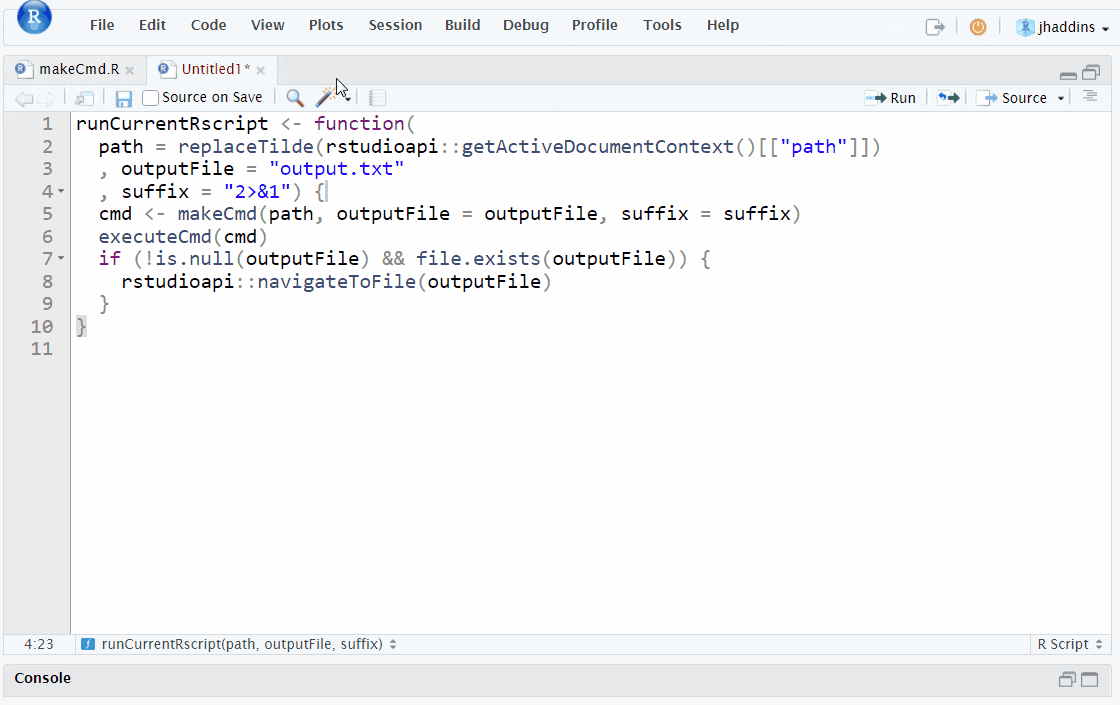
RStudio Addins Part -2

# Quick intro to documentation with roxygen2

## 1. Documenting your first function

To help us generate documentation easily we will be using the roxygen2 package. You can install it using install.packages("roxygen2"). Roxygen2 works with in-code tags and will generate R’s documentation format .Rd files, create a NAMESPACE, and manage the Collate field in DESCRIPTION (not relevant to us at this point) automatically for our package.

Documenting a function works in 2 simple steps:

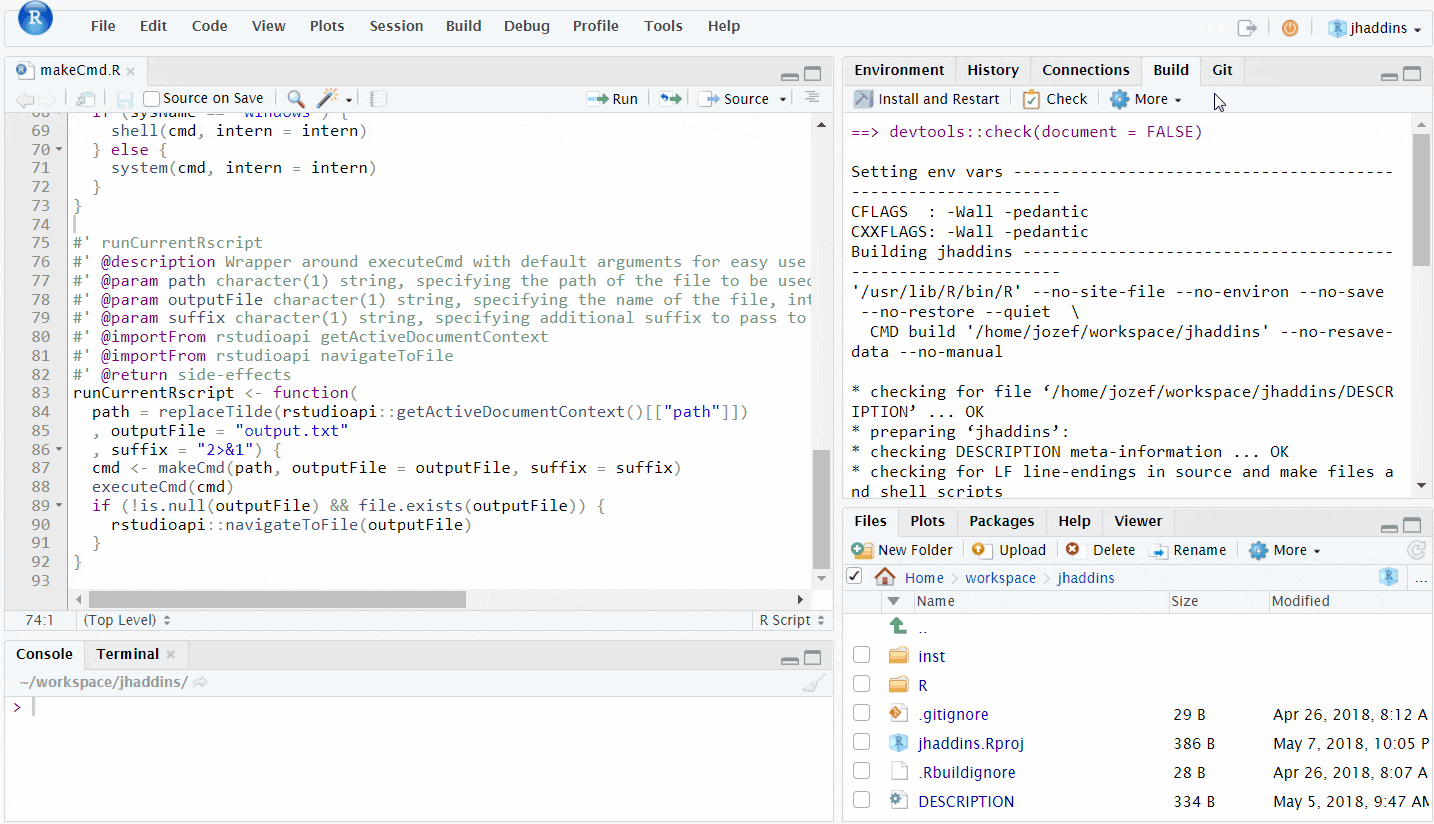


Documenting a function

1. Inserting a skeleton - Do this by placing your cursor anywhere in the function you want to document and click Code Tools -> Insert Roxygen Skeleton (default keyboard shortcut Ctrl+Shift+Alt+R).
2. Populating the skeleton with relevant information. A few important tags are:

* #' @params - describing the arguments of the function
* #' @return - describing what the function returns
* #' @importFrom package function - in case your function uses a function from a different package Roxygen will automatically add it to the NAMESPACE
* #' @export - if case you want the function to be exported (mainly for use by other packages)
* #' @examples - showing how to use the function in practice

## 2. Generating and viewing the documentation



Generating and viewing the documentation

1. We generate the documentation files using roxygen2::roxygenise() or devtools::document() (default keyboard shortcut Ctrl+Shift+D)
2. Re-installing the package (default keyboard shortcut Ctrl+Shift+B)
3. Viewing the documentation for a function using ?functioname e.g. ?mean, or placing cursor on a function name and pressing F1 in RStudio - this will open the Viewer pane with the help for that function

## 3. A real-life example

Let us now document runCurrentRscript a little bit:

#' runCurrentRscript

#' @description Wrapper around executeCmd with default arguments for easy use as an RStudio addin

#' @param path character(1) string, specifying the path of the file to be used as Rscript argument (ideally a path to an R script)

#' @param outputFile character(1) string, specifying the name of the file, into which the output produced by running the Rscript will be written

#' @param suffix character(1) string, specifying additional suffix to pass to the command

#' @importFrom rstudioapi getActiveDocumentContext

#' @importFrom rstudioapi navigateToFile

#' @seealso executeCmd

#' @return side-effects

runCurrentRscript <- function(

path = replaceTilde(rstudioapi::getActiveDocumentContext()[["path"]])

, outputFile = "output.txt"

, suffix = "2>&1") {

cmd <- makeCmd(path, outputFile = outputFile, suffix = suffix)

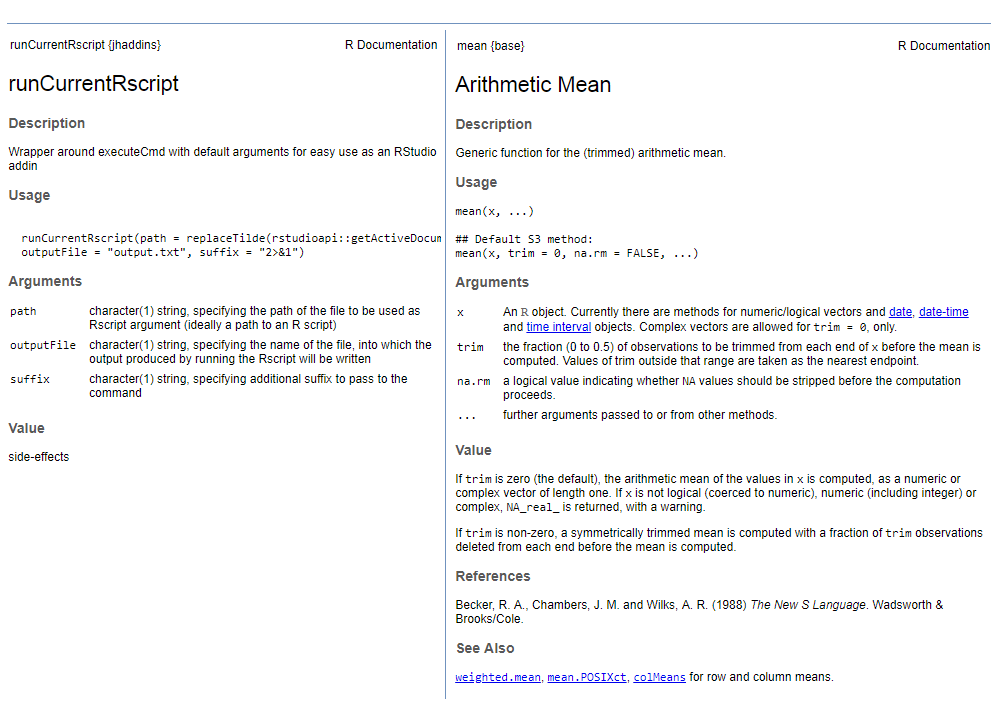
executeCmd(cmd)

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

rstudioapi::navigateToFile(outputFile)

}

}

As we can see by looking at ?runCurrentRscript versus ?mean, our documentation does not quite look up to par with documentation for other functions:

What is missing if we abstract from the richness of the content is the usage of markup commands (tags) for formatting and linking our documentation. Some of the very useful such tags are for example:

* \code{}, \strong{}, \emph{} for font style
* \link{}, \href{}, \url{} for linking to other parts of the documentation or external resources
* \enumerate{}, \itemize{}, \tabular{} for using lists and tables
* \eqn{}, \deqn{} for mathematical expressions such as equations etc.

For the full list of options regarding text formatting, linking and more see Writing R Extensions’ Rd format chapter

# Our addins to make documenting a breeze

As you can imagine, typing the markup commands in full all the time is quite tedious. The goal of our new addin will therefore be to make this process efficient using keyboard shortcuts - just select a text and our addin will place the desired tags around it. For this time, we will be satisfied with simple 1 line tags.

## 1. Add a selected tag around a character string

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")

}

## 2. Apply the tag on a selection in an active document in RStudio

We will make the functionality available for multi-selections as well by lapply-ing over the selection elements retrieved from the active document in RStudio.

addRoxytag <- function(tag = NULL) {

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))

}

## 3. Wrappers around addRoxytag to be used as addin for some useful tags

addRoxytagCode <- function() {

addRoxytag(tag = "code")

}

addRoxytagLink <- function() {

addRoxytag(tag = "link")

}

addRoxytagEqn <- function() {

addRoxytag(tag = "eqn")

}

## 4. Add the addin bindings into addins.dcf and assign keyboard shortcuts

As the final step, we need to add the bindings for our new addins to the inst/rstudio/addins.dcf file and re-install the package.

Name: addRoxytagCode

Description: Adds roxgen tag code to current selections in the active RStudio document

Binding: addRoxytagCode

Interactive: false

Name: addRoxytagLink

Description: Adds roxgen tag link to current selections in the active RStudio document

Binding: addRoxytagLink

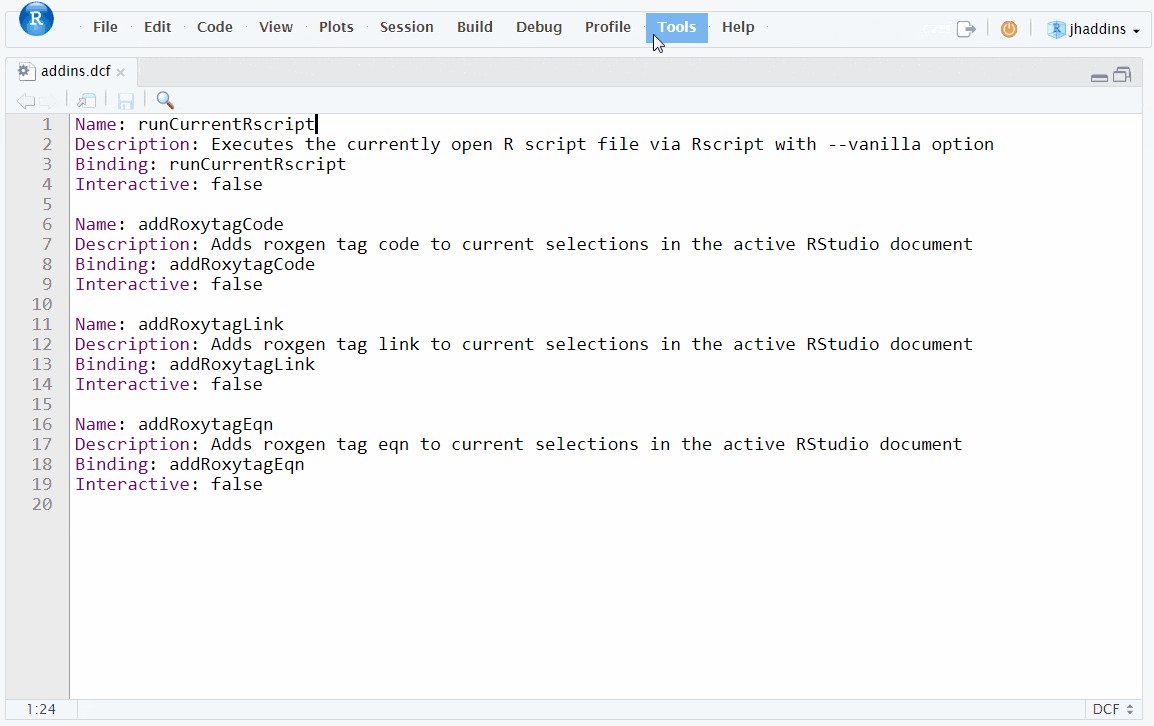
Interactive: false

Name: addRoxytagEqn

Description: Adds roxgen tag eqn to current selections in the active RStudio document

Binding: addRoxytagEqn

Interactive: false



assigning keyboard shortcuts to addins

# The addins in action

And now, let’s just select the text we want to format and watch our addins do the work for us! Then document the package, re-install it and view the improved help for our functions:

