# R Intermediate Tips and Tricks

# 1. Shortcuts ------------------------------------------------------------

Use Ctr Shift R to create sections and more easily navigate through code.

Use Ctr I for proper indentation.

Use Ctr Shift M to insert a pipe %>%

if(TRUE){

print("start")

df <- data.frame(a = c(1,2), b = c("a", "b")) %>%

dplyr::mutate(a = a+1,

c = a/2)

print("end")

}

# 2. Parameters -----------------------------------------------------------

Parameterise your code wherever possible and define them at the top of the script

overwrite\_outputs = TRUE

# read data ---------------------------------------------------------------

test\_df <- data.frame(x = c(1,2,NA, 4), y = c(NA,NA,NA, NA), z = c("a","b", NA, NA))

names(test\_df)

names(test\_df)[1] <- id\_col

test\_df

# write data --------------------------------------------------------------

if(overwrite\_outputs){

print("saving data")

readr::write\_csv(test\_df, "test\_df.csv",)}

# 3. Using colSums, rowSums, colMeans to deal with NA values ---------------

count\_na <- function(df) {

print(df %>% is.na() %>% colSums())

}

percentag\_na <- function(df) {

print(df %>% is.na() %>% colMeans())

}

count\_na(airquality)

percentag\_na(airquality)

# drop rows with only missing values & drop cols with only missing values

drop\_allNA\_rows\_cols <- function(df) {

df[rowSums(is.na(df)) !=ncol(df), colSums(is.na(df)) < nrow(df)]

}

Tidyverse way:

dat %>% filter(if\_any(everything(), ~ !is.na(.)))

# 4. Using pacman package for installing/loading packages -----------------

Try pacman for easy package installation and loading in one step.

a) you can list them all in one set of brackets,

b) pacman loads a package if it already is installed and installs it if not, all in one

pacman::p\_load(dplyr, sf)

# 5. Easily accessing previously run code in console ----------------------

In R Studio if you want to re-run a previously run line of code, type the first (few) letters and then press Ctr and the up arrow to go through previously run commands that start with the same characters.

# 6. Get matches from partial strings and multiple patterns ----------------------

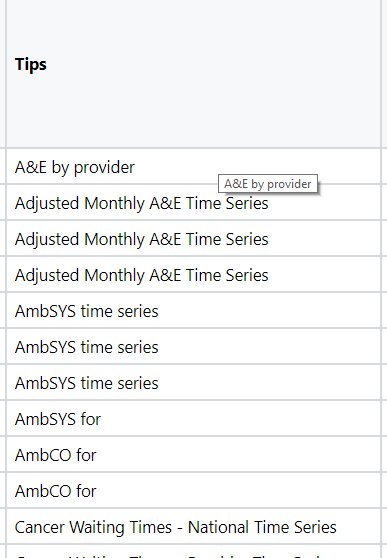
#first list all the files you are working with

xlFiles <- list.files(path = xldirs, pattern = "xls\*", include.dirs = TRUE)

relFiles <- c() #then create an empty variable to store each filename that matches

#have a vector of keywords that you want use to pick up files

For example, I have a dataframe which I use as my lookup file that has this column of keywords:



#next match your files using the keywords column

for(i in table\_row){ #loop to append each matched filename to relFiles

relFiles[i] <- grep(paste0(variables$Tips[i], collapse = "|"),

xlFiles, ignore.case=TRUE, value = TRUE) }

# 7. Write out multiple excel files at the same time ----------------------

Put your dataframes in a list to export them out to one excel file

Then use a loop to keep the names of your dataframes as sheet names in your excel file

#writing excel workbook

for (i in 1:length(tidydata)){

openxlsx::write.xlsx(tidydata, file = paste0('./Output/', "testing ", Sys.Date(),".xlsx"),

asTable = TRUE,

sheetNames = names(tidydata[i]) #tidydata is a list of 27 dataframes

)

}

# 7. Non standard evaluation and ! ! bang bang ----------------------

Parts of R utilise Non-standard evaluation (NSE), which roughly lets you modify an expression or its meaning after it has been issued but before it is executed.

# For instance: subset(mtcars, hp > 250)

# the function subset interrupts hp > 250 before it is run, instead it looks for a column called hp in mtcars rather than look for an object hp in the workspace. If we were to use standard evaluation it would be: mtcars[mtcars$hp > 250,]

## NSE in tidyverse

# When we use the tidyverse, we directly refer to column names rather than directly referencing the column within an object (using NSE)

mtcars %>% mutate(NewColumn = 2\*mpg)

## A problem encountered in using NSE in dplyr - functions

# Here is a function that creates a count of the number of rows for a grouping variable

group\_func <- function(grouping\_var){

data %>% group\_by(grouping\_var) %>% summarise(n()) }

# If we try use that function on a small table of favourite foods and colours

data <- data.frame(matrix(c("Fish fingers","Burgers","Purple","Green"),ncol=2,byrow = TRUE))

names(data) <- c("Favourite food","Favourite Colour")

group\_func(`Favourite food`)

# So why doesn't this work?

# When we run the function R is searching for 'grouping\_var' within it's scope and not searching for `Favourite food`, it is quoting the argument 'grouping\_var' and not finding it.

# When it is quoted rather than evaluated that means the function is using NSE.

# Some loose definitions

# 'Evaluated' argument - obeys R's usual evaluation rules. R passes arguments by value, they are evaluated in the calling environment and the values are passed to the function

# 'Quoted' argument - To capture an unevaluated expression - these are captured by a function and processed in a different way. As we saw with subset(mtcars, hp > 250), hp is being captured by the function and processed to search for that column name in mtcars.

# 'Unquotation - To be able to selectively evaluate parts of an otherwise quoted expression. Unquoting a single expression will evaluate it, and inline the results.

***# So how do we resolve our problem?***

***# 1. We have to quote our argument***

***# 2. We have to tell dplyr it is already quoted, which is done by unquoting it***

# The bang-bang !! operator in rlang will unquote it for us

group\_func\_fixed <- function(grouping\_var){

data %>% group\_by(!!grouping\_var) %>% summarise(n())}

group\_func\_fixed(quo(`Favourite food`))

# Now `Favourite food` isn't being searching for in the global environment, it holds evaluation. It is now being quoted and unquoted in the function.

# 8. For loops ----------------------

# Outside of functions we may come into an issue with say a for loop

for(i in names(data)){

data %>% group\_by(i) %>% summarise(n()) %>% print()}

# This time we have gone a further step back, we are now inputting a string, and so we need to convert it to a symbol as that is what dplyr expects.

# Using rlang::sym we can do this

names(data)

rlang::sym(names(data)[1])

# Now using what we had before we can fix the for loop

for(i in names(data)){

symbol\_i <- rlang::sym(i)

data %>% group\_by(!!symbol\_i) %>% summarise(n()) %>% print()}

# 9. Tic Tok package ----------------------

In the past when trying to determine how long bits of code was taking to run, and work out where the bottle necks were i'd used an obvious but unelegant solution;

start.time <- Sys.time()

Sys.sleep(1)

Sys.time() - start.time

# This is time consuming to run and gives a minimal amount of information. The tictoc package is a simple alternative to this method.

# The tictoc package contains functions that allow you to set off timers and end times using the tic() and toc() function. Upon the tic receive it's complimentary toc

# the time taken will display. The functions can also be nested, you can set off 3 tics in a row followed by 3 tocs

library(tictoc)

tic.clear() # Clears any unclosed tics

tic.clearlog() # Clear any tictocs that have been added to the log.

tic(msg = "My first tic completed") # Set off a 'start.time', it will also display the message it's paired toc() is run

tic(msg = "My second tic completed") # Setting off a second start.time, the first tic has not been ended yet. The next toc will now end this tic

toc(log = TRUE) # Will send the most recent tic and submit that time to a log

toc(log = TRUE) # Will end the first tic and submit that time to a log

tic.log() # We can see all the timings here, in a long script this can be useful for evaluation