Statistical Publication Automation Guide

Introduction

Note that this is not intended as a comprehensive guide to programming with R or the full suite of pckages available. This is merely meant as an introduction to the basic open-source tools available to government statisticians looking to automate statistical publications. Read through this guide and see what aspects of the RAP toolkit would be best suited for your project and follow the given links to understand how you may use them to a greater degree.

Set Up

To start with, to execute R code within your markdown document, you enter it in the following format:

# Within this you may run functions and import files. This code wi, by default, execute whenever you knit the document. If you wish it to NOT run then include the following additional arguments:  
  
# {r, eval = FALSE}  
  
# By default, markdown will display these code chunks in your final document. If you wish them to be hidden, enter the following additional argument:  
  
# {r, echo = FALSE}

A full walkthrough on the various r setup commands can be found here: <https://bookdown.org/yihui/rmarkdown/r-code.html>.

We will use the in-house data sets of iris and mtcars for most of this walkthrough but I will include sample code below showing how to read in multiple files that fit a particular pattern from a given directory.

If you are working from a particular in-house server for security purposes, such as your H drive, be sure to set this as your working directory to avoid accidental writing to the laptop hard drive:

setwd <- "H://s023456/Department/Folder"

If you are handling sensitive data then you can customise RStudio to stop saving your history to your computer within the Global Options dropdown. To do this, click on the following tabs:

Tools > Global Options > General > untick(Always save history) > Apply

Outside of this, take care not to handle sensitive data in an irresponsible manner, exactly the same as you would with any other project.

Importing Data

You can read in files directly from an SQL database, using the odbc package and save them to the folder we have set as our working directory. From here we may use the read\_csv or read\_excel commands to bring them into RStudio, manipulating the tables as we do so. If you are reading in csv files then use the following functions:

# Read\_csv function works as following:  
  
read\_csv("Directory/Folder/File.csv")  
  
# Read\_csv can be given supplimentary arguments. For example:  
  
# Trim\_ws = TRUE will trim off empty whitespace from the original document  
  
# Doing this manually for all files is time consuming so we can tell R to search for and import all files that match a particular pattern. For example, this would tell R to grab all files of a csv format:  
  
data\_path = getwd()  
  
files <- dir(data\_path, pattern = "\*.csv")  
  
data <- files %>%  
 map(~ read\_csv(file.path(data\_path, .)))  
  
# While the following would grab all files that have a particular date in the title:  
  
data\_path = getwd()  
  
files <- dir(data\_path, pattern = "\*2019")  
  
data <- files %>%  
 map(~ read\_csv(file.path(data\_path, .)))

The full extent of the read\_csv function can be studied here: <https://readr.tidyverse.org/reference/read_delim.html>.

If you are looking to read in excel files then you will need the readxl package. Note: there is a package that is meant to be used for both reading and writing excel files, called xlsx, but it will not work on SCOTS due to Java incompatability so you’ll need to do this the following way:

barley <- read\_excel("H:/Directory/Folder/File.xlsx", sheet=1, range = cell\_rows(1:2))  
  
# The sheet argument allows us to specify a particular sheet to read in, the default is 1  
  
# Range allows us to read in a particular section. So for example, range = sheet1!B2:G16 will only read in data from cells B2 to G16 in sheet 1

More read\_excel examples can be found here: <https://readxl.tidyverse.org>.

We will look at the package writexl later in this guide.

Data Maipulation with Tidyverse

Once we have the data read into the RStudio environment, we can used the tidyverse package to manipulate the tables into a format better suited to our analysis and output. Note: Tidyverse is simply a collection of the most useful data manipulation packages, such as stringr and dplyr, in one single package.

The full extent of data manipulation tools available within the tidyverse suite of packages is beyond the scope of this guide but if you are interested in more advanced tidyverse training, consult with your line manager about Jumping Rivers who regularly provide training to gov employees in the Glasgow and Edinburgh areas.

Basic Styling

Markdown, providing we are knitting to html, supports html script outside of our code blocks. In this manner, we can colour and adapt our text the same way we would use utilise CSS for webpage design. I am afraid you cannot use this background shading if you are producing a word document. Shading is acheived with the style parameters. Define your div class then enter your chosen text / code chunk within a subsequent div:

## mpg cyl disp hp drat wt qsec vs am gear carb  
## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4  
## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4  
## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1  
## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1  
## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

## [1] "text text text text"

Table Styling

Tables can be made more attractive and stylised using the kableExtra package. This format can be copied for insertion into excel documents if needed. Below is an example output using the mtcars dataframe. A tutorial on the full extent of kables capabilities can be found here: <https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html>

mtcars[1:5, ] %>%  
 kable() %>%  
 kable\_styling()

mpg

cyl

disp

hp

drat

wt

qsec

vs

am

gear

carb

Mazda RX4

21.0

6

160

110

3.90

2.620

16.46

0

1

4

4

Mazda RX4 Wag

21.0

6

160

110

3.90

2.875

17.02

0

1

4

4

Datsun 710

22.8

4

108

93

3.85

2.320

18.61

1

1

4

1

Hornet 4 Drive

21.4

6

258

110

3.08

3.215

19.44

1

0

3

1

Hornet Sportabout

18.7

8

360

175

3.15

3.440

17.02

0

0

3

2

Alternatively, you could try displaying with formattable. A comprehensive walkthrough on the use of the formattable package can be found here: <https://www.littlemissdata.com/blog/prettytables>. Formattable may not seem very different but it can be used for some attractive visuals such as the table below where each cell will darken with a greater value:

mtcars[1:5, ] %>%  
 formattable()

mpg

cyl

disp

hp

drat

wt

qsec

vs

am

gear

carb

Mazda RX4

21.0

6

160

110

3.90

2.620

16.46

0

1

4

4

Mazda RX4 Wag

21.0

6

160

110

3.90

2.875

17.02

0

1

4

4

Datsun 710

22.8

4

108

93

3.85

2.320

18.61

1

1

4

1

Hornet 4 Drive

21.4

6

258

110

3.08

3.215

19.44

1

0

3

1

Hornet Sportabout

18.7

8

360

175

3.15

3.440

17.02

0

0

3

2

customGreen0 = "#DeF7E9"  
customGreen = "#71CA97"  
formattable(mtcars[1:5, ], align =c("l","c","c","c","c", "c", "c", "c", "c", "c", "c", "c"), list(  
 ` ` = formatter("span", style = ~ style(color = "grey",font.weight = "bold")),   
 `mpg`= color\_tile(customGreen0, customGreen),  
 `cyl`= color\_tile(customGreen0, customGreen),  
 `disp`= color\_tile(customGreen0, customGreen),  
 `hp`= color\_tile(customGreen0, customGreen),  
 `drat`= color\_tile(customGreen0, customGreen),  
 `wt`= color\_tile(customGreen0, customGreen),   
 `qsec`= color\_tile(customGreen0, customGreen),  
 `vs`= color\_tile(customGreen0, customGreen),  
 `am`= color\_tile(customGreen0, customGreen),  
 `gear`= color\_tile(customGreen0, customGreen),  
 `carb`= color\_tile(customGreen0, customGreen)  
))

mpg

cyl

disp

hp

drat

wt

qsec

vs

am

gear

carb

Mazda RX4

21.0

6

160

110

3.90

2.620

16.46

0

1

4

4

Mazda RX4 Wag

21.0

6

160

110

3.90

2.875

17.02

0

1

4

4

Datsun 710

22.8

4

108

93

3.85

2.320

18.61

1

1

4

1

Hornet 4 Drive

21.4

6

258

110

3.08

3.215

19.44

1

0

3

1

Hornet Sportabout

18.7

8

360

175

3.15

3.440

17.02

0

0

3

2

Graphical Output

The most capable and accessible graphical output packages are plotly and ggplot2. Plotly is best suited for attractive, colourful and reactive output while ggplot has more advanced features and allows high-level customisation. Plotly is ideal for dashboards for example, while ggplot is better suited for academic statistical models. For a more thorough walkthrough of and examples of how to output different types of graphs on plotly, visit: <https://plot.ly/r/>

This colourful output demonstrates how plotly graphs can be designed with just a few lines of code:

trace\_0 <- rnorm(100, mean = 5)  
trace\_1 <- rnorm(100, mean = 0)  
trace\_2 <- rnorm(100, mean = -5)  
x <- c(1:100)  
  
data <- data.frame(x, trace\_0, trace\_1, trace\_2)  
  
p <- plot\_ly(data, x = ~x, y = ~trace\_0, name = 'trace 0', type = 'scatter', mode = 'lines') %>%  
 add\_trace(y = ~trace\_1, name = 'trace 1', mode = 'lines+markers') %>%  
 add\_trace(y = ~trace\_2, name = 'trace 2', mode = 'markers')  
  
p

## PhantomJS not found. You can install it with webshot::install\_phantomjs(). If it is installed, please make sure the phantomjs executable can be found via the PATH variable.

Unit Testing

One of the advantages of automation is to build in testing at every step of your programme to automatically check for undesireable values in your code, such as duplicates or missing strings. The two most commonly used packages are testthat and futile.logger.

In their simplest format, the syntax of a testthat function must be constructed with two arguments: our expected output and our actual output.

string <- "Testing is fun!"  
  
expect\_match(string, "Testing is fun!")

If the test passes, and in this case it will as the strings match, then your markdown document will knit and the functions can be executed without issue. However, try changing the second argument in the expect\_match function. If the test fails then an error message will be displayed and the markdown document will not knit. If you need to check that your functions or scirpt is outputting a particular value then this package is ideal. Testthat has many different functions for more customised testing, a complete walkthorugh can be found at: <http://r-pkgs.had.co.nz/tests.html>.

In the context of automating statistical publications, another useful testing function is duplicated. With this function you can identify and remove identical values within vectors or columns. Here is an example using the iris dataframe to identify, extract and remove rows with duplicate values in the Petal.Length column:

table <- as\_tibble(iris[1:5, ])  
  
table

## # A tibble: 5 x 5  
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## <dbl> <dbl> <dbl> <dbl> <fct>   
## 1 5.1 3.5 1.4 0.2 setosa   
## 2 4.9 3 1.4 0.2 setosa   
## 3 4.7 3.2 1.3 0.2 setosa   
## 4 4.6 3.1 1.5 0.2 setosa   
## 5 5 3.6 1.4 0.2 setosa

# Test for duplicate values - evaluates to true or false  
duplicated(table$Petal.Length)

## [1] FALSE TRUE FALSE FALSE TRUE

# Extract duplicate values  
extract\_duplicates <- table$Petal.Length[duplicated(table$Petal.Length)]  
  
extract\_duplicates

## [1] 1.4 1.4

# Remove rows with duplicate values from original table  
table <- table[!duplicated(table$Petal.Length), ]  
  
table

## # A tibble: 3 x 5  
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## <dbl> <dbl> <dbl> <dbl> <fct>   
## 1 5.1 3.5 1.4 0.2 setosa   
## 2 4.7 3.2 1.3 0.2 setosa   
## 3 4.6 3.1 1.5 0.2 setosa

Another testing option is the futile.logger package. This is the unit testing package used in the Udemy RAP tutorial. It’s strength is that it allows you to customise error messages produced when tests fail, meaking it easier to identify exactly what is going wrong with your data, especially if you have multiple tests. The syntax utilises conditional logic, asking the code to execute a particular command if a set of circumstances is met.

Good practise is to create a section of your code dedicated to testing the underlying data and work through the required tests in a methodical manner using the mtcars dataframe:

# Checks  
futile.logger::flog.info("Initiating checks on dataframe object to ensure data is as expected")

## INFO [2019-10-24 20:45:56] Initiating checks on dataframe object to ensure data is as expected

# This test should pass  
futile.logger::flog.debug("Checking that our object is a dataframe...")  
 if (!is.data.frame(mtcars))  
 {  
 futile.logger::flog.error("The object must be a dataframe", mtcars, capture = TRUE)  
 }  
  
# This test should fail  
futile.logger::flog.debug("Checking there are only the 3 columns we require...")  
 if (length(colnames(mtcars)) > 3)  
 {  
 futile.logger::flog.error("Your dataframe has more than 3 columns.")  
 }

## ERROR [2019-10-24 20:45:56] Your dataframe has more than 3 columns.

# This test should pass  
futile.logger::flog.debug("Checking there are no missing values...")  
 if (anyNA(mtcars)) stop("There cannot be missing values in your dataframe")

Note that failing tests will then publish their particular error message to your console window and appear in your markdown r script. This will help pinpoint exactly why your data isn’t looking the way you want or why your build isn’t completing.

Unit testing can be done to any degree you wish. We recommend building it in at stages where large amounts of data tidying and manipulation is taking place.

String Interpolation

String interpolation is simply telling your markdown document to publish values that are reactive and based on your underlying output. This can allows you to automate the inclusion summary statistics and highlights, taking a lot of the leg work out of your final write-up. The two most commonly used functions for this are paste0 and glue.

Let’s use these functions to produce highlights of the iris data set, note that they require different syntax but achieve largely the same result:

# The glue function from the glue package  
greatest\_value <- max(iris$Petal.Length)  
glue("The greatest petal length in the dataset is {greatest\_value}")

## The greatest petal length in the dataset is 6.9

smallest\_value <- min(iris$Petal.Width)  
paste0("The smallest petal width is ", smallest\_value)

## [1] "The smallest petal width is 0.1"

This can also be used in a more direct manner to call particular cell values:

glue("The first value of petal width within the dataset is {iris$Petal.Width[1]}")

## The first value of petal width within the dataset is 0.2

Exporting Data

Exporting data in R can be in csv or excel format quite easily. Once again, you will need to specify the working directory to write to if yo uare working with any form of sensitive data. you will need to utilise the writexl and readr packages for each respectively.

write\_csv(data, "Directory/Folder/File.csv", append = FALSE, col\_names = !append)  
  
write\_xlsx(data, path = "Directory/Folder/File"(fileext = ".xlsx"), col\_names = TRUE,  
 format\_headers = TRUE)

In both cases, the initial command will create a new file. Using the append argument, you may append additional sheets to an existing file. For example, if you had three tables and wished to save them each as a sheet in the same file, you would pass these tables to the function in the form of a list:

write\_xlsx(c(table1, table2, table3), "Directory/Folder/File.xlsx", col\_names = TRUE,  
 format\_headers = TRUE)