

Writing Documents Programatically in R

Liam Wright

14 June, 2019

Introduction

- Microsoft Office is good, but it relies heavily on manual action and switching between programmes.
- Manual actions aren't **reproducible**, which is **inefficient** if you need to repeat something; **risky** if you are doing anything complex; and **lacking transparency** if you want to share your work.
- We'll be talking about packages within **R** that allow you to download, clean, and analyse data and to report the results all using code within the **RStudio** interface.

An Introduction to R and RStudio

- R is an open source statistical package, which has a steeper learning curve than other programmes (e.g. Stata or SAS), but has a lot more capability.
- R has a very active online community of users, who can help solve any problems you have.
- Alone, R does not have a Graphical User Interface (GUI), so most people also download RStudio. This makes it easier to see your data and results in one place, do certain point-and-click actions, etc.
- RStudio comes with a number of packages which enable you to produce documents programatically in R.

An Introduction to R and RStudio

- R can work with data from many different file formats, including `.xlsx`, `json`, `xml`, `.dta`, `.sas`, etc.
- R allows you to have multiple datasets in memory at once (unlike Stata). This can make the code quite verbose.
- Data is stored in *objects*. There are several types of object in R, including vectors, matrices, n-dimensional arrays, lists, and data frames. Data frames are like datasets in Stata or tables in SAS.

An Introduction to R and RStudio

- R doesn't have much functionality built in. Instead, extra functionality is introduced by installing **packages**. Packages are set of functions.
- Packages only need to be installed once, but have to be loaded into memory in each session.

An Introduction to R and RStudio

- The basic R syntax is:

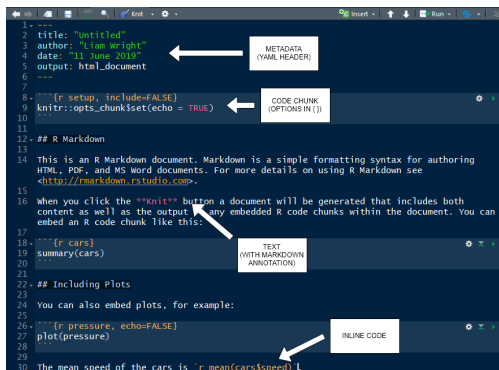
```
object <- function(argument=value,...)
```

- where <- is the **assignment** operator.
- = is also a valid assignment operator, but is used less due to its use in setting function arguments.

- To create documents in R, we use a package called `rmarkdown` which is bundled with RStudio.
- `rmarkdown` allows you to produce documents from plain text files which have three components:
 - Metadata (YAML Header)
 - Text (Annotated with Markdown)
 - Code (Inline or Chunks)
- The plain text files are **compiled** into final documents with formatting rendered and R code correctly and stastical output “knitted” in.

rmarkdown

- To open your first R Markdown file, click File > New File > R Markdown...
- The file that opens includes a simple tutorial on R Markdown.



The screenshot shows the RStudio editor with an R Markdown file. The code is as follows:

```
1- ---
2- title: "Untitled"
3- author: "Liam Wright"
4- date: "11 June 2019"
5- output: html_document
6- ---
7-
8- ```{r setup, include=FALSE}
9- knitr::opts_chunk$set(echo = TRUE)
10- ```
11-
12- ## R Markdown
13-
14- This is an R Markdown document. Markdown is a simple formatting syntax for authoring
15- HTML, PDF, and MS Word documents. For more details on using R Markdown see
16- <http://rmarkdown.rstudio.com>.
17-
18- When you click the **Knit** button a document will be generated that includes both
19- content as well as the output of any embedded R code chunks within the document. You can
20- embed an R code chunk like this:
21-
22- ```{r cars}
23- summary(cars)
24- ```
25-
26- ## Including Plots
27-
28- You can also embed plots, for example:
29-
30- ```{r pressure, echo=FALSE}
31- plot(pressure)
32- ```
33-
34- The mean speed of the cars is `r mean(cars$speed)`.
```

Annotations with arrows point to the following parts of the code:

- METADATA (YAML HEADER)**: Points to the first six lines (lines 1-6).
- CODE CHUNK (OPTIONS IN {})**: Points to the opening and closing backticks of the first code chunk (lines 8-10).
- TEXT (WITH MARKDOWN ANNOTATION)**: Points to the paragraph of text between lines 14 and 20.
- INLINE CODE**: Points to the inline R code ``r mean(cars$speed)`` at the end of line 34.

Figure 1: Simple R Markdown Document

Metadata

- Document metadata is stored in the **YAML header**.
- The metadata determines the format of the document, including the file type to be produced (e.g. .pdf, .docx, slides, article), its title, the authors, etc.
- An important component of this is the “output:” argument. This is used to specify formatting options important for a particular file type.
- YAML is a **markup** language (more on markup soon). Note, indentation is important in YAML.

- The text sections are written in a simple **markup** language called **Markdown** (specifically, Pandoc's Markdown).
- A markup language is one in which formatting is specified by adding annotations (code) to the plain text file. The formatting is only rendered when the plain text file is **compiled** into a final document (e.g a .pdf).
- In Markdown, a header is specified with #, while **bold face** is specified by enclosing some text with two asterisks either side (i.e., ****this would be bold face****).

- Markup languages are distinguished from “What You See is What You Get” languages/text editors (e.g. Word). In these formatting is rendered directly in the editable document.
- In Markdown, images are added to the document via hyperlink to the relevant file `*![Caption](filepath.png)`
 - Images therefore have to be saved separately from the plain text file.
- For an introduction to the Markdown syntax, visit **StackEdit.io**.

- `rmarkdown` uses **Pandoc** to compile the final document. Pandoc comes bundled with RStudio.
- For some file types, additional software may be required to compile the document.
- For `.pdf`'s on a local installation of R, a distribution of LaTeX (e.g. MiKTeX) is needed for compilation. LaTeX is another widely-used markup language.
- To render `.pdf`'s correctly some LaTeX packages may need to be specified in the YAML header.

- Code sections are used to demarcate operations to be run in R or another statistical language compatible with `rmarkdown` (e.g. Python, Stata, SAS).
- Code is delineated with leading and trailing backticks. Leading backticks are followed by a letter denoting the program to run the code in. For code chunks, three backticks are used, while for inline code, a single backtick is used.
- The code chunks are run and then “knitted” in to the document.

Code



Figure 2: Backtick

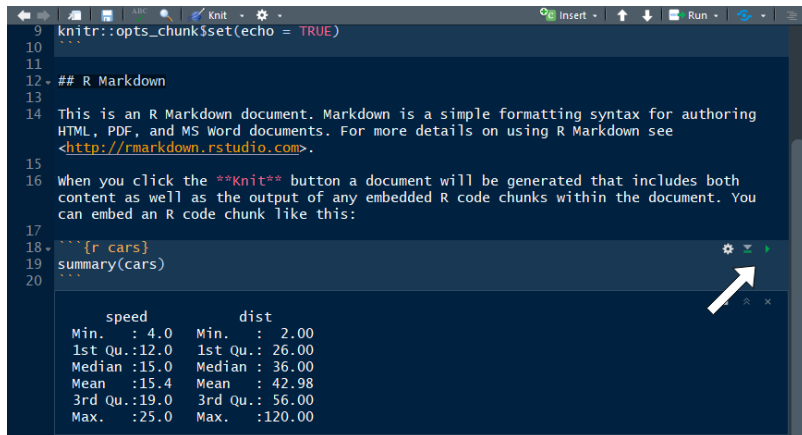
- With code chunks, chunk options are specified within `{}` braces following the leading backticks. These options include:
 - The program to run the code in (required),
 - A name for the code chunk,
 - Whether the code should “echoed” and/or output “hidden”
 - Other formatting options - for instance, figure dimensions.
- Global options - which set chunk defaults - can be set using the following function in a code chunk:

```
knitr::opts_chunk$set(option = value,...)
```

- where `option` is an argument to be set at `value`.
- Note, `knitr` is a package `rmarkdown` uses to create R Markdown documents.
- (The `::` operator enures the `opts_chunk$set` function is used from the `knitr` package.)

Code

- Code chunks can be run without the full document being compiled by clicking the green arrow next to the chunk. This helps for checking everything is working without running the whole analysis.



```
9 knitr::opts_chunk$set(echo = TRUE)
10 ...
11
12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring
15 HTML, PDF, and MS Word documents. For more details on using R Markdown see
16 <http://rmarkdown.rstudio.com>.
17
18 When you click the Knit button a document will be generated that includes both
19 content as well as the output of any embedded R code chunks within the document. You
20 can embed an R code chunk like this:
```

```
18 {r cars}
19 summary(cars)
20 ...
```

speed	dist
Min. : 4.0	Min. : 2.00
1st Qu.:12.0	1st Qu.: 26.00
Median :15.0	Median : 36.00
Mean :15.4	Mean : 42.98
3rd Qu.:19.0	3rd Qu.: 56.00
Max. :25.0	Max. :120.00

- Inline code can be used to place results directly into the main text.
- With conditional statements (if conditions), you can write text programmatically depending on the results of your analysis.
- Below is an example of some conditional text written in R Markdown.

```
Total employment currently stands at `r format(total.emp.current, scientific=FALSE,
big.mark=",")` thousand individuals. Employment has `r
if(total.emp.change>0){"increased"}` `r if(total.emp.change<0){"decreased"}` year on
year by `r abs(total.emp.change)` thousand individuals. Total employment is at its
highest level since `r total.emp.prevhighest`. Year on year, employment has `r
sentence`.
```

Figure 4: Conditional Text using Inline Code

Producing R Markdown Documents

- Once we have finished our plain text file, which contains all the text, code and formatting we want, we click the `Knit` button in RStudio.
- Clicking the arrow next to `Knit` opens up more file format options. Choosing one of these will automatically add the relevant metadata to the `YAML Header` to produce the file correctly.

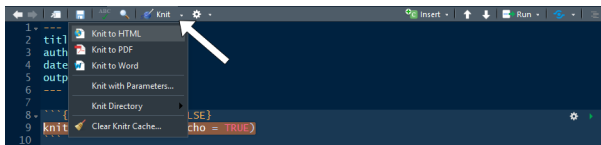


Figure 5: Knit Button

Producing R Markdown Documents

- Clicking the Knit button creates an independent R session.
 - Data in memory from the existing session will not be available when running the code chunks, so will need to be reloaded or created again in the script itself.
- This isn't the case if you call `rmarkdown` in the console directly (using the `render` function), but is the more sensible option to ensure your document is fully **reproducible** - that is, it stands alone.

```
render("My R Markdown.Rmd", pdf_document())
```

Accessing Data Programmatically

- To increase reproducibility, you can even use R to download your data directly from the internet.
- There are two ways of doing this:
 - Application Programming Interfaces (**APIs**)
 - **Web Scraping**

APIs

- In very simplistic terms, an API is a protocol for sending requests to a server and getting **predictable** information back. (This **YouTube video** is helpful)
- In the basic form we will use, an API is a URL. Think of a URL as an instruction to a server (e.g the BBC) to return a specific document (a news article).
- Usually a URL will return a HTML document (<https://www.bbc.co.uk/news>), which your web browser renders as a nice looking webpage.
- With the APIs we are interested in, a piece of structure text will be returned instead, with specific content determined by the **API Call** (the parameters of the URL string) and format following rules set by the API owner (and typically published online).

- The resulting structured text can be parsed into a dataset because it is predictable.
- Typical formats are JSON and xml, which structure data into (nested) attribute-value pairs.
- Below is the JSON text returned by the **Postcodes.io API** for SW1H 9NA (Caxton House):

```
{"status":200,"result":{"postcode":"SW1H 9NA","quality":1,"eastings":529805,"northings":179570,"country":"England","nhs_ha":"London","longitude":-0.13132,"latitude":51.500171,"european_electoral_region":"London","primary_care_trust":"Westminster","region":"London","lsoa":"Westminster 020A","msoa":"Westminster 020","incode":"9NA","outcode":"SW1H","parliamentary_constituency":"Cities of London and Westminster","admin_district":"Westminster","parish":"Westminster, unparished area","admin_county":null,"admin_ward":"St James's","ced":null,"ccg":"NHS Central London (Westminster)","nuts":"Westminster","codes":{"admin_district":"E09000033","admin_county":"E99999999","admin_ward":"E05000644","parish":"E43000236","parliamentary_constituency":"E14000639","ccg":"
```

Figure 6: <http://api.postcodes.io/postcodes/SW1H9NA>

ONS API

- The ONS has a number of APIs for exploring and downloading its time series data directly.
- The format of the API Call for downloading time series data is:
 - https://api.ons.gov.uk/dataset/%7Bdataset_id%7D/timeseries/%7Btimeseries_id%7D/data
- Dataset and time series IDs can be found by looking in spreadsheets, using the ONS's **Time-Series Explorer**, or using the ONS's **Data Explorer APIs**.
 - The time series ID for labour market statistics is LMS.
- Note, not all of the ONS's data series are available via the API.

Using APIs in R

- To use APIs in R, we use multiple R packages: `httr`, `jsonlite`, `lubridate`, and `tidyverse`.
- In the following code, we download total workforce numbers using the ONS API:

```
path <- "dataset/LMS/timeseries/MGTM/data"
raw <- GET(url = "https://api.ons.gov.uk", path = path)
return <- rawToChar(raw$content) %>% fromJSON()
df <- return$quarters
```


Using APIs in R

- GET (from `httr`) returns the JSON data. It is named after the **HTTP “GET” protocol** for communicating with a server over the web.
 - The specific protocol you need to use will be stated in the API documentation.
 - Another commonly used protocol is POST, which is used to send data to a server and is typically used for longer API calls.

```
path <- "dataset/LMS/timeseries/MGTM/data"
raw <- GET(url = "https://api.ons.gov.uk", path = path)
return <- rawToChar(raw$content) %>% fromJSON()
df <- return$quarters
```

Using APIs in R

- The returned JSON file from GET is in unicode. `rawToChar` converts this into human readable form.
- `fromJSON()` creates a list object from this data. A list in R is a collection of objects that can be of different types.
 - The `%>%` operator is from the `tidyverse`. It passes the resulting object from the function on the left and places it into the first argument of the function on the right.

```
path <- "dataset/LMS/timeseries/MGTM/data"
raw <- GET(url = "https://api.ons.gov.uk", path = path)
return <- rawToChar(raw$content) %>% fromJSON()
df <- return$quarters
```

Using APIs in R

- The quarters object in return (accessed via the \$ operator) is a data frame containing quarterly information.
- With these four lines, I've extracted the data I need straight from the internet.

```
path <- "dataset/LMS/timeseries/MGTM/data"  
raw <- GET(url = "https://api.ons.gov.uk", path = path)  
return <- rawToChar(raw$content) %>% fromJSON()  
df <- return$quarters
```

Using APIs in R

- Because the API parameters are predictable, I can write code to **loop** over multiple API Calls.
 - Below, the `assign` function creates an object which has the name stored in `i` (i.e. `MGTM`, `JF6H`,...).

```
ids <- c("MGTM", "JF6H", "EQ4U", "EQ4W",  
        "JF6F", "EQ4Q", "EQ4S")  
  
for (i in ids){  
  path <- paste("dataset/LMS/timeseries/", i, "/data")  
  raw <- GET(url = "https://api.ons.gov.uk", path = path)  
  return <- rawToChar(raw$content) %>% fromJSON()  
  assign(i, return$quarters)  
}
```

Other Useful APIs

- There are lots of data providers who have APIs.
- These include:
 - Google Maps
 - NomisWeb (see the `nomisr` package)
 - StatXplore
 - Public Health England Fingertips (see the `fingertipsR` package)

Web Scraping in R

- Information available via API is preferable as the returned data follows particular rules.
- Where APIs aren't available, you can use R to scrape the data from the internet instead.
- This works best where websites have predictable URLs (e.g. stable) and the format of the webpage is predictable too.

Web Scraping in R

- For instance, we can use R to find the location of the “current” spreadsheet for a particular data series on the ONS website.
 - More concretely, we are interested in finding the URL underlying the xls (637.4 kB) hyperlink below.

The screenshot shows the ONS website page for the dataset 'JOBS05: Workforce jobs by region and industry'. The page includes a breadcrumb trail: Home > Employment and labour market > People in work > Employment and employee types > JOBS05: Workforce jobs by region and industry. Below the dataset name, there is a section for 'About this dataset' and 'Your download options'. Under 'Your download options', there are two links: 'Current' and 'xls (637.4 kB)'. A black arrow points to the 'xls (637.4 kB)' link. To the right of the download options, there is a section for 'Contact details for this dataset' which lists Mark Williams, labour.market@ons.gov.uk, and Telephone: +44 (0)1633 456728. There is also a button 'View all data related to Employment and employee types'.

Figure 7: ONS Hyperlink

Web Scraping in R

- We can scrape web data in R using the `rvest` package. Because we are downloading an Excel spreadsheet, we need the `xlsx` package, too.
- To first find the webpage element we want to scrape, though, we need to download the **Selector Gadget** extension in Chrome.
 - This tool generates a “CSS selector” for an element in a webpage. The CSS selector is used by `rvest` to extract specific information from the HTML document.

Web Scraping in R

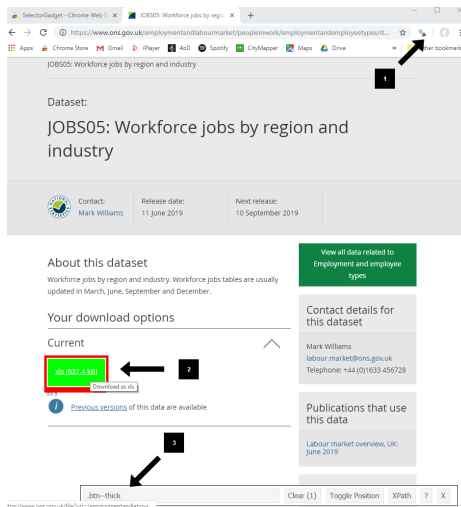


Figure 8: Using Selector Gadget in Chrome

Web Scraping in R

- Once we have the CSS selector, we can then write the following code. This returns the website that the current xls file is stored at.

```
ons <- "https://www.ons.gov.uk"
url <- paste0(ons,
              "/employmentandlabourmarket/peopleinwork",
              "/employmentandemployeetypes/datasets",
              "/workforcejobsbyregionandindustryjobs05")
current <- read_html(url) %>%
  html_nodes('.btn--thick') %>%
  html_attr('href') %>%
  paste0(ons,.)
```

Web Scraping in R

- `read_html()` downloads the html document from `url`.
`html_nodes()` extracts node information at the specified CSS selector position (`.btn--thick`). `html_attr('href')` extracts the hyperlink from this node.

```
ons <- "https://www.ons.gov.uk"
url <- paste0(ons,
              "/employmentandlabourmarket/peopleinwork",
              "/employmentandemployeetypes/datasets",
              "/workforcejobsbyregionandindustryjobs05")
current <- read_html(url) %>%
  html_nodes('.btn--thick') %>%
  html_attr('href') %>%
  paste0(ons,.)
```

Web Scraping in R

- Getting this right is a matter of trial and error.
- Note, in the final line, “.” is used alongside the %>% operator to place the object from the left hand side into the second argument of the paste0 function.

```
ons <- "https://www.ons.gov.uk"
url <- paste0(ons,
              "/employmentandlabourmarket/peopleinwork",
              "/employmentandemployeetypes/datasets",
              "/workforcejobsbyregionandindustryjobs05")
current <- read_html(url) %>%
  html_nodes('.btn--thick') %>%
  html_attr('href') %>%
  paste0(ons,.)
```

Web Scraping in R

- Now we have the spreadsheet URL, we can download it to a temporary file, open the sheets we want, clean them and analyse the resulting data.

```
tmp <- tempfile(fileext = ".xls")  
download.file(url = current, destfile = tmp, mode="wb")  
sheet <- read.xlsx(tmp,sheetName="1. North East")
```

Web Scraping in R

- Note, in practice this will be difficult to do programmatically.
- Spreadsheets generally don't follow strict rules, so can change arbitrarily through time (particularly the case with the ONS)
- There is often issues of reading in Excel data in the correct format - e.g. dates.
- In this case, it is still beneficial to use R to create reports, but fully **automating** the process will not be possible.

Concluding Remarks

- We have shown how documents can be made programatically using R.
- For routing reports, these could be fully updated using APIs/Web Scraping and conditional statements at each new data release with no further human intervention.
- In practice, though, writing code for every eventually may be too time-consuming and will break if there is any change in the structure of the data.
- Reproducibility principles can be used for some elements of your report (loading data creating tables, graphs, etc.) with others left flexible.
- What is most sensible will be context specific.

Further Reading

- For more information on R Markdown, see the free online **bookdown** textbook.
- There are several free online R courses. See, for instance, the **Datacamp** website and **Roger Peng's Coursera** course.
- For intermediate or advanced users interested in using R for all aspects of data science from data cleaning to analysis, see Hadley Wickham's free online textbooks **R for Data Science** and **Advanced R**.
- **R for Data Science** gives an extensive overview of the **tidyverse** package, which is a widely used set of functions and principles for data science in R.

Further Reading

- For those interested in creating graphs, see documentation on the `ggplot2` package, including Kieran Healy's wonderful free online textbook **Data Visualization**.
- More free R textbooks are listed on the **bookdown** website.
- RStudio also produce a number of helpful **cheatsheets**.
- For advice on specific questions, see forums on **StackOverflow**.
- To look at others' code see **GitHub** and the **Open Science Foundation**.