JUMPING RIVERS

AUTOMATED REPORTING

JUMPINGRIVERS.COM

Contents

1
R markdown and "knitr"

What is markdown?

Markdown is the name given to a lightweight markup language designed to have plain, simple syntax that can be converted[1] to another format, `html` or `pdf` say. There are a number of similar markdown languages, when you edit a github page or answer a stackoverflow question, you are using a form of markdown. The emphasis of these sorts of languages is on speed and simplicity of writing.

Although simple, it is very powerful. This book https://csgillespie.github.io/efficientR/ was written using markdown!

[1] Or marked up.

R markdown (.Rmd)

R markdown is a flavour of markdown that allows R code to be run as part of the mark up process. It allows a combination of LaTeX expressions, eg. math environment, with code chunks but with much less verbose syntax than a full latex document. The consequence of this is such that writing a document is quick but some flexibility and polish is lost.

R markdown is based on Pandoc which is a type of markdown aimed at being flexible in it's output type. Originally markdown was invented for `html` shorthand.

There is a very nice cheat sheet from RStudio

http://shiny.rstudio.com/articles/rm-cheatsheet.html

that summarises the various markdown syntax elements but here are a few elements to get started. This chapter will cover the basics of markdown.

R markdown is not unique, markdown languages have been around since 2004 with the goal of being easily readable with simple syntax. Markdown documents are then marked up to the desired document type, eg. HTML.

The `rticles` R package contains a number of templates for R markdown including some which conform to specific journal formatting requirements.

Header

All R markdown files start with a header that looks similar to this[2]:

```
---
title: "My title"
author: "Me"
date: "15 September 2015"
output: html_document
---
```

[2] This is known as a YAML header. YAML stands for *YAML Ain't Markup Language* and is a human readable data serialisation commonly used for configuration files.

The first three items are obvious. The final line contains the output type. R markdown allows a number of output types

- `html_document` – mark up to `html` code, for documents to be viewed in a web browser.
- `pdf_document` – marks up to LaTeX code then compiles as a pdf[3].
- `word_document` – marks up to create a document in the style of Microsoft word (preview requires Microsoft word/ libre office).
- `ioslides_presentation` – a html format for presentations.
- `beamer_presentation` – a beamer presentation as a pdf.

[3] On Windows, you need to install miktex, `http://miktex.org/`

Knitr files can also be used to create presentations in the same way you might create a presentation from LaTeX, beamer say.

The beauty of markdown is that it is really easy! For example, starting a line with '#' will create a level 1 header, surrounding text with '**' will make the text **bold**. See the cheat sheet for a full list.

Exercise: Complete exercise 1 in the `chapter1` vignette.

```
vignette("chapter1", package="jrShiny")
```

Embedded R code with `knitr`

`knitr`[4], pronounced "knitter" is a general purpose package for dynamic report generation in R. It supports multiple file types including markdown and LaTeX. It is particularly nice because it allows R code and text to be written together in a single document. R code is embedded in *chunks* which are then processed by the `knitr` package.

[4] `https://github.com/yihui/knitr`

Figure 1.1: The `knitr` logo.

Chunks

Chunks in R markdown look like

````
```{r}
## R code
```
````

where everything between the sets of back-ticks is regular R code to be evaluated. For example

````
```{r}
 1 + 1
 x = 3^2
 x
```
````

results in the following:

```
1 + 1

## [1] 2

x = 3^2
x

## [1] 9
```

Option name	Value	Effect
`eval`	logical	Whether to evaluate a code chunk
`echo`	logical	Whether to include the source code in output
`include`	logical	Whether to include the chunk results in output
`fig.width`, fig.height	numeric	Dimensions of plots in inches

Table 1.1: Some useful `knitr` chunk options.

- R code is treated exactly the same as using the normal R GUI interpreter.
- Variables can be used to store results.
- Graphs can be created.
- Use routines from various packages.

It is possible use other languages in code chunks. For example, change `r` to `python` and you can run python code!

R CODE CHUNKS can also take a host of options. These are specified within the curly braces after the `r`, separated by commas.

```
```{r, option1, option2}```
```

Table 1.1 gives some standard options[5].

[5] `http://yihui.name/knitr/options` gives a full list of `knitr` chunk options.

Inline R code

We often want to include the result of R expressions within a sentence or line of text. This is referred to as an inline expression. The syntax for this is simple, we just enclose the R expression, in the middle of a sentence with `` `r ...R code... ` ``. As an example

The character used here is a back-tick, not a quote. On my keyboard this is to below Esc.

> The result of $5 * 6$ is `` `r 5*6 ` ``.

prints

> The result of $5 * 6$ is 30.

This sort of inline R code would typically be used for short calculations, or a reference to a variable that has been stored elsewhere in the script. For larger sections of code we use *chunks*.

Processing a .Rmd file

Using RStudio makes processing a R markdown document very easy. At the top of the text editor window is the button `Knit HTML`. If you are not in RStudio then the function `render` in the `rmarkdown` package does the same.

If you like keyboard shortcuts, then Ctrl + Shift + K does the same.

You will need to make sure you have the latest version of pandoc installed for this.

```
rmarkdown::render("file.Rmd")
```

EXERCISE: Complete exercise 2 in the `chapter1` vignette.
```

```
vignette("chapter1", package="jrShiny")
```

Hyper Text Markup Language

Hyper Text Markup Language `HTML` is a markup language for describing web pages; it is one of the foundations of the internet. Whenever you go to a webpage, you are downloading an `HTML` document that your web-browser renders. An `HTML` document is just a collection of tags:

```
<tag>Some text</tag>
```

Each tag describes how the text should be formatted.

We translate the markdown file into HTML. For example, the markdown statement `**bold**` is converted into HTML statement `<strong>bold</strong>`.

Exercise: Complete exercise 3 in the `chapter1` vignette.

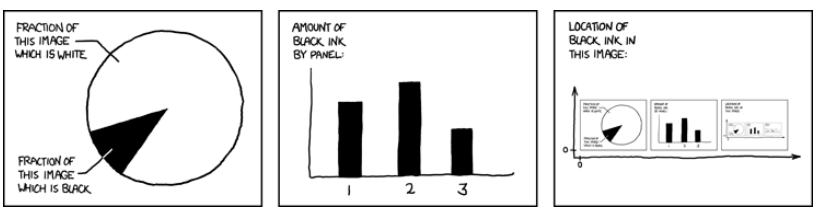

Figure 1.2: https://xkcd.com/688/

2
Using htmlwidgets

The `htmlwidgets`[1] package provides an easy to use framework for creating R bindings[2] to JavaScript libraries. The resultant widgets can be

- Used just like conventional R plots;
- Included in R Markdown documents;
- Incorporated into shiny web applications[3]
- Saved as standalone web pages.

Using an existing R htmlwidget requires very little code.

[1] `http://www.htmlwidgets.org/`

[2] interface

[3] The crucial difference between shiny and an htmlwidget, is that a widget is run *client* side; shiny is server side.

Exercise: Complete exercise 1 in the `chapter2` vignette:

```
vignette("chapter2", package="jrShiny")
```

Creating widgets is also straightforward[4]. In this course we won't discuss creating our own widget, but if you are interested in the process see

`http://www.htmlwidgets.org/develop_intro.html`

New HTML widgets are being released all the time. The course package provides a list of the currently available widgets.

Exercise: Complete exercise 2 in the `chapter2` vignette.

[4] In fact someone created a new widget every week for a year `http://www.buildingwidgets.com/`

Figure 2.1: `https://xkcd.com/937/`

3
Flexdashboard layouts

In this chapter we are going to look at the very new `flexdashboard` package[1]. This package is used to layout groups of related graphics as a dashboard. A dashboard can include a variety of components such as

[1] http://rmarkdown.rstudio.com/flexdashboard/

- Interactive JavaScript data visualisations based on htmlwidgets[2].
- R graphical output including base, lattice, and grid graphics.
- Tabular data (with optional sorting, filtering, and paging).
- Value boxes for highlighting important summary data.
- Gauges for displaying values on a meter within a specified range.
- Text annotations of various kinds.

[2] The previous chapter.

There are number of options that allow components to be organised on multiple pages. There is also a mobile device option. Interactivity can be included using `htmlwidgets` or Shiny to build fully custom interactions in R.

Layouts

The `flexdashboard` package makes it easy to create dashboards.

- Single column (fill and scrolling);
- Rows;
- Multiple columns;
- Tabs;
- Pages.

New chart panes are created using ### and new columns (or rows) are created using `----------`).

For example, to create the layout in figure 3.1, is just a simple markdown document

```
---
title: "Single Column (Fill)"
output:
  flexdashboard::flex_dashboard:
    vertical_layout: fill
---
### Chart 1
```{r}
plot(rnorm(10))
```

Chart 2
```{r}
plot(rnorm(10))
```
```

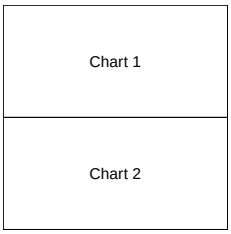

Figure 3.1: Column layout

EXERCISE: Complete exercise 1 in the `chapter3` vignette.

Sizing

We can also change the sizes of dashboard panes. If no size attributes are specified then each chart size is relative to it's `knitr` figure size[3].

To specify the size, we use the `data-width` and `data-height` attributes. These attributes establish the *relative* size between charts laid out in the same dimension.

[3] 6 × 4.8 inches or 576 × 460 pixels by default.

EXERCISE: Complete exercise 2 in the `chapter3` vignette.

Appearance

You can alter the overall theme of the dashboard by adding a theme element in the header. You can also add bespoke `css` code, logos and favicon's. See

`http://rmarkdown.rstudio.com/flexdashboard/using.html#appearance`

for more details.

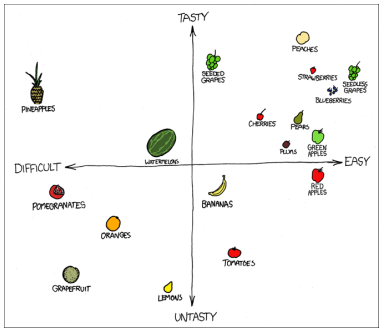

Figure 3.2: `https://xkcd.com/388/`

4
Parameterised reports

Including parameters

R Markdown documents can optionally include one or more parameters. Parameters are useful when you want to re–render the same report with distinct values for various key inputs, for example:

- Running a report specific to a department or geographic region.
- Running a report that covers a specific period of time.
- Running mulitple versions of a report for distinct sets of core assumptions.

Declare them in the YAML

R Markdown parameter names, types and default values are all declared in the YAML header section at the top of a document. Parameters are declared using the `params` field, for example:

```
---
title: My Document
output: html_document
params:
  region: east
---
```

This tells us that we have a single parameter called region which takes a default value "east".

The standard R data types can be dealt with in this way.[1] Alternatively arbitrary R objects can be used by specifying the value using an R expression. For example to specify a data you could use:

[1] Supported standard data types are `character`, `integer`, `numeric` and `logical`.

```
---
title: My Document
output: html_document
params:
  start: !r as.Date("2015-01-01")
---
```

Note that R expressions are prefaced with `!r` to indicate that the value is an R expression rather than a literal value.

Using Parameters

The declared parameters are automatically made available within the knitted environment as components of a list names `params`. We could access the start value from the previous example in a chunk with the following R code

```
```{r}
params$start
```
```

Whenever you knit the document to be previewed in RStudio (or otherwise call the `rmarkdown::render()` function with no arguments) the default parameter values listed in the YAML header will be used.

Passing alternative parameter values

To vary the parameters in the document use the `params` argument of the `rmarkdown::render()` function.

```
rmarkdown::render("MyDocument.Rmd",
        params = list(
                region = "west",
                start = as.Date("1970-01-01")
        )
)
```

User interface for parameters

The previous section describes invoking a report using the `rmarkdown::render()` function to allow customization of it's parameters. It is also possible to provide a user–interface for specifying those parameters.

If your document contains parameters and you specify `params = "ask"` argument in the call to `render()` then a user interface is provided to specify parameter values. For example, consider:

```
---
title: "My Document"
output: html_document
params:
  minimum: 100
  region: east
  data: results.csv
---
```

If you call the `render()` function as

```
rmarkdown::render("MyDocument.Rmd", params = "ask")
```

you will get a basic user interface for parameter entry.

The user interface is generated using `shiny`. We haven't really talked about `shiny`, but it is a R package that allows us to create interactivity with our graphics and documents. As a result however we can take advantage of standard `shiny` input controls. You can customise inputs by adding the appropriate values to the parameter YAML. For example we could customise the parameters in the previous example to use the `shiny` `sliderInput` `selectInput` and `fileInput` controls:

```
---
title: "My Document"
output: html_document
params:
  minimum:
    label: "Minimum:"
```

```
    value: 100
    input: slider
    min: 0
    max: 1000
  region:
    label: "Region:"
    value: east
    input: select
    choices: [east, west, north, south]
  data:
    label: "Input dataset:"
    value: results.csv
    input: file
---
```

The type of input control is specified by the `input` field. The currently supported types are shown in table 4.1.

Input type	Shiny Function
checkbox	checkboxInput
numeric	numericInput
slider	sliderInput
date	dateInput
text	textInput
file	fileInput
radio	radioButtons
select	selectInput

Table 4.1: Shiny input options for parameterised reports.

Task Scheduling

It is also possible to schedule R scripts/processes with the Windows task scheduler. This allows Windows users to automate R processes at specific timepoints from R itself using the R package `taskscheduleR`. This package is not currently available on CRAN, however can be installed directly from github if you have the `devtools` package.

```
devtools::install_github("bnosac/taskscheduleR")
```

Example usage

With `taskscheduleR` you can

- Get the list of scheduled tasks.
- Remove a task.
- Add a task
  - A task is basically a script with R code which will be run using Rscript.
  - You can schedule tasks 'ONCE', 'MONTHLY', 'WEEKLY', 'DAILY', 'HOURLY', 'MINUTE', 'ONLOGON', 'ONIDLE'.
  - The task log contains the stdout and stderr (standard output and errors) which was run at that timepoint. This log can be found in the same folder as the R script.

To create a new task we could use the following R code:

```
library(taskscheduleR)
myscript = "scripttorundaily.R"
taskscheduler_create(taskname = "namefortask",
                     rscript = myscript,
                     schedule = "DAILY",
                     starttime = "09:10",
                     startdata = format(Sys.Date() + 1, "%d/%m/%Y"))
```

Which would run the script found in a file called `scripttorundaily.R` daily at 9:10 am starting tomorrow. Some other examples of using the task scheduler are shown below:

```
## Run every week on Sunday at 09:10
taskscheduler_create(taskname = "myfancyscript_sun",
                     rscript = myscript,
                     schedule = "WEEKLY",
                     starttime = "09:10",
                     days = "SUN")

## Run every 5 minutes, starting from 10:40
taskscheduler_create(taskname = "myfancyscript_5min",
                     rscript = myscript,
                     schedule = "MINUTE",
                     starttime = "10:40",
                     modifier = 5)
```

Viewing and removing tasks

To view all currently scheduled tasks:

```
tasks = taskscheduler_ls()
str(tasks)
```

Or to delete tasks:

```
taskscheduler_delete(taskname = "namefortask")
```

The package also contains an RStudio addin which gives a GUI within RStudio for generating tasks. For this to work you will also need the `miniUI` and `shiny` packages.

Note that to schedule tasks it is likely that you will need administrator rights on the machine.

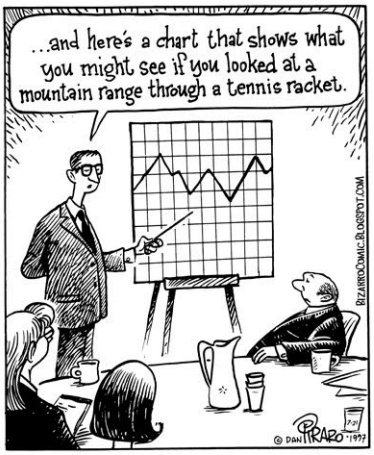

Figure 4.1: `http://www.bizarrocomics.com/`

Deployment

If your document does not contain any shiny components, then you can just email the html file. However, if there are any shiny elements, there are two options.

1. Set up your own shiny server[2]. There is an open-source version and a paid version.
2. Host your document or app in the cloud with `shinyapps.io`. Again there are free and paid for versions.

Full details on uploading your app are at

[2] `https://www.rstudio.com/products/shiny/shiny-server/`

http://shiny.rstudio.com/articles/shinyapps.html

Once you have created a shiny apps account, to deploy your document or app, just run

```
library("rsconnect")
deployApp("name_of_app.Rmd")
```

The first time I uploaded an app it took approximately 20 minutes, but subsequent uploads have been quicker.

Appendix

Course R package

This course has an associated R package. Installing this package is straightforward. First install the `drat` package

```
install.packages("drat")
```

Then run the command[3]

```
drat::addRepo("jr-packages")
```

Then install the package as usual

```
install.packages("jrShiny")
```

To load the package, use

```
library("jrShiny")
```

[3] This adds a new repo URL to you list of repositories.