Writing R Notebooks

Version 2020-04

Licence

This manual is © 2019-20, Simon Andrews.

This manual is distributed under the creative commons Attribution-Non-Commercial-Share Alike 2.0 licence. This means that you are free:

- to copy, distribute, display, and perform the work
- to make derivative works

Under the following conditions:

- Attribution. You must give the original author credit.
- Non-Commercial. You may not use this work for commercial purposes.
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a licence identical to this one.

Please note that:

- For any reuse or distribution, you must make clear to others the licence terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Full details of this licence can be found at
http://creativecommons.org/licenses/by-nc-sa/2.0/uk/legalcode

Contents

INTRODUCTION 4

BACKGROUND 4
NOTEBOOK CONCEPTS 4
STARTING A NOTEBOOK 5

EDITING YOUR NOTEBOOK 7

HEADER 7
Table of Contents 7
Rendering of tibbles and data frames 9
MARKDOWN 9
Headings 9
Simple text formatting 10
Lists 10
Tables 11
Fixed width text and Quotes 12
Escaping 12
R CODE 13
Inserting a new code chunk 13
Writing R code 13
Running R code 13
Breaking code into chunks 14
Naming chunks 15

TWEAKING CODE CHUNKS 16

DIFFERENCES TO CORE R CODE 16
Handling of the working directory 16
Drawing multi-layered plots 16
ADJUSTING GRAPHICAL OUTPUT 18
Changing the size of the graphics pane 18
Changing the format of graphical output 19

RENDERING YOUR NOTEBOOK 21

Introduction

R notebooks are an integrated script type in RStudio. They have a number of advantages over traditional text script files and are the ideal way to record interactive analyses of your data.

This booklet provides some instructions for how to use the R Notebook system to create professional looking reports from your R analysis. It assumes that you're familiar with how to write R within a standard script file and it won't go into any detail about writing R – just the additional parts you need to include it as part of a notebook.

Background

Traditionally R code has been written in a plain text file, either using a standard text editor, or within the context of a development environment such as RStudio. Writing R in this way means that you have a complete record of the actions you took to perform your analysis, and by passing the file to the RScript program you can easily rerun a whole analysis.

Text based scripts have a number of disadvantages though.

1. They separate the output from the code. In a traditional R environment the code is in the script file. Text output goes to the console and graphical output either goes to separate files, or opens in a new window away from the code. There is no easy way to associate the outputs of the analysis with the specific parts of the script which generated them.

2. It's difficult to know that they're complete. Part of the attractiveness of a scripted environment is that it is a way to completely reproduce what you did. When running scripts though it is easy to end up running things out of order, or accidentally changing or deleting parts of your code. It can be difficult to know at the end that you have a complete record of what you did. Even re-running the entire analysis can be confounded by what data you have loaded already, and cleaning your environment can inhibit debugging if something is missing.

3. They don't encourage comment. Whilst scripts are a good way of documenting *what* you did, they aren't very good at allowing you to explain *why* you did it, or what you think the results mean. You can, of course, add comments to an R program by using the # symbol, but this isn't the most friendly system in the world. In the end your commentary around the code is just as important as the details of what you did – especially if your code is being shared with others.

4. They're not very pretty. Script files only contain plain ASCII text. They have no capacity for formatting or presentation of your comments or results – they're not very aesthetically pleasing.

R Notebooks are a way to attempt to address all of these problems.

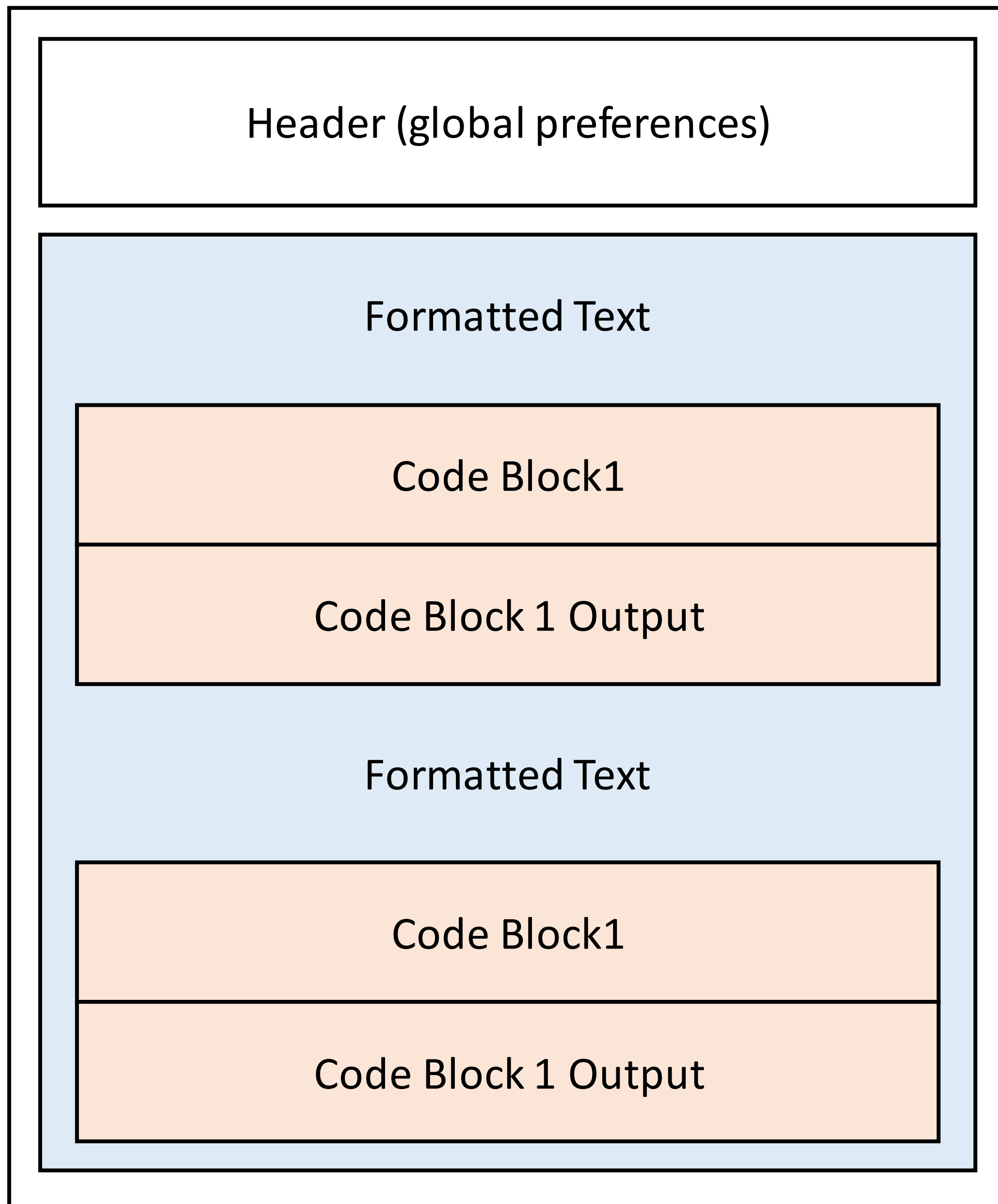
Notebook Concepts

Notebooks are an additional document type which you can use to write R code. Within RStudio you can launch a notebook by selecting

```
File > New File > R Notebook
```

Notebooks rely on a number of R packages to be able to work. The first time you open one you will be prompted to install the necessary packages. Once that's done then you will see a basic template open.

The basic structure of a notebook looks like this:



So the document is divided into sections. Apart from a header which has to come first the remaining document is by default a type of formatted text called 'markdown'. Within this text document you can then embed blocks of R code, which you can then run. The output of the code then becomes embedded into the document immediately below the code which generated it. The documents have systems to more nicely represent some of the common data types such as data frames and tibbles.

Notebooks are written as a live, interactive document where you can both edit the blocks and run the code to update the output in real time. Once your notebook is complete though you can then pass it through a renderer to generate a final, fully formatted document in HTML, PDF or MS Word format. This formatted document can be passed as a report to others and represents an attractive final version of the analysis.

Starting a notebook

Notebooks are a standard document type available in RStudio. You can start a notebook by selecting File > New File > R Notebook. The first time you open a notebook you may be prompted to install a set of R packages required to generate notebooks. These should be installed automatically once you agree.

The new notebook document will open to reveal a short template document illustrating the capabilities of the notebook. You can use this as a starting point to write your own document.

Since some properties of the notebook are defined by where it is saved in the filesystem it is a good idea to immediately save the new document. If you are using the notebook to process data files then it is conventional to save the document in the same directory as the data. Notebooks must be saved as files with a .Rmd file extension so they are recognised correctly by RStudio. Using a different file extension will cause them to stop working.

Editing your Notebook

Editing the notebook is done via changing the text in the template document which is created when you start a new notebook. Below we will go through the options you have to alter the information in the different sections of the file.

Header

The notebook header is at the start of the document. It is started and ended with a set of three minus signs on one line. The standard template should show a complete header which you can then modify. The initial header template looks like:

```
---
title: "R Notebook"
output: html_notebook
---
```

Sections in the header are keywords followed by a colon. Initially only a title and output section will be defined. The title is easily changed, but more interesting things can be done with the output section.

The output section determines what format of output is generated by the document when it is compiled. Standard formats for the output are:

- HTML document
- PDF document
- Word document

A single document can be compiled to any or all of these and each time a new format is selected then the header will be extended to include it.

For each document type you can then specify additional options which will determine how the document is presented. Adding a new document type will generate an option which looks like:

```
---
title: "R Notebook"
output:
  html_document:
    df_print: paged
---
```

Since the most common output format for these notebooks is HTML we're going to focus on the options for that format.

Table of Contents

If in your main markdown document you have used headings then you can collate this information together into a table of contents which can then be placed at the start of the document. Notebooks support multi-level headings, and you can choose how many levels you wish to collate into the table of contents.

An example simple toc statement is shown below along with the table which is generated.

```
---
title: "Table of Contents"
output:
  html_document:
```

```
    toc: true
    toc_depth: 2
---
```

Table of Contents

- Introduction
  - What is this document
  - Why am I writing it
  - What does it show
- Methods
- Results
- Conclusion

The default `toc_depth` is 3 so this is slightly more summarised than you'd normally get.

The table of contents is placed at the top of the document and links to the corresponding sections below. One issue with this is that whilst it makes it easy to navigate from the top of the document to a specific section, it is more difficult to then jump around inside the document.

To make this easier an additional option is to 'float' the table of contents so that it always sits in a fixed position on the left of the document. This makes navigation easier. This style of TOC also dynamically expands and contracts sub-sections so that it stays a manageable size. To enable this feature you add the `toc_float: true` option to the `html_document`.

```
---
title: "Table of Contents"
output:
  html_document:
    toc: true
    toc_float: true
    toc_depth: 3
---
```

- Introduction
  - What is this document
    - What sort of document
    - Where is it saved
  - Why am I writing it
  - What does it show
- Methods
- Results
- Conclusion

Table of Contents

Introduction

What is this document

What sort of document

Different document types

Meanings of different documents

A related parameter to the inclusion of a table of contents is the addition of automated section numbers. This option causes automatically incrementing numbers of the form [section] . [subsection] . [sub-subsection] to be added to each of the rendered headings, as well as to the table of contents. You can add this by including the parameter: `number_sections: true` to the `html_document` options.

1 Introduction
1.1 What is this document
1.2 Why am I writing it
1.3 What does it show
2 Methods
3 Results
4 Conclusion

Table of Contents

1 Introduction

1.1 What is this document

1.1.1 What sort of document

1.1.1.1 Different document types

1.1.1.2 Meanings of different documents

Rendering of tibbles and data frames

Another option for `html_document` is how tabular data is represented in the document. By default the usual text representation of tibbles and data frames is replaced by a rendered HTML equivalent where a nicely styled HTML table is used and supplemented with controls to access columns or rows which don't fit into the available space.

The rendering of tabular data is controlled by the parameter `df_print`. The valid options are:

Option value	Text or HTML	Fits to space	Restricts rows	Paging controls
`default`	text	no	no	no
`kable`	HTML	no	no	no
`tibble`	text	yes	yes	no
`paged`	HTML	yes	yes	yes

Only works on data frames. Tibbles act like the tibble option

The standard template included `df_print: paged` and this is the default if you don't specify anything (rather than `default`!). Generally the paged structure works well, but you might want to use `tibble` in cases where the HTML version omits some useful information (total dimensions or grouping information for example).

Markdown

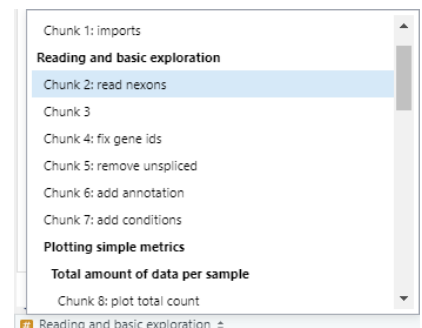
The default format for the text you put into a Notebook is 'Markdown'. This is a simple and intuitive text formatting language which allows you to make headings, lists, tables and other nicely rendered features, but is still readable in its raw state. For the most part markdown looks like normal plain text, but by knowing how it works you can create nicely structured and rendered reports.

When you're actually working in your notebook you will see the raw markdown code. It's only when you compile the document to produce the final report that you'll see the rendered versions of the elements you've created.

We'll go through some of the main types of markdown you'll ever use.

Headings

The most important piece of markdown you'll use are headings. These not only help to break up the text in your document but they provide important points of reference in your document. Headings are used for creating your table of contents (if you've opted to include one), and are also visible at the bottom of the document in a little menu, which is a really convenient way to jump around to different parts of your notebook.



Notebooks support multiple levels of headings. You can have up to 6 levels of heading in your document where level 1 is the most important and level 6 is the least important.

Headings can be created in two different ways. The first is to simply underline the heading text by putting a row of either equals signs (like a double underline) or minus signs (a single underline) on the line beneath.

```
A level 1 heading
=================

A level 2 heading
-----------------
```

The number of underlines doesn't matter – it doesn't have to match the length of the heading text, but it's visually nicer if it does.

This method only works for level 1 and 2 headings. An alternative which works for any level of heading is to prefix the heading text with a number of hash symbols (###) equal to the level of the heading.

```
# A level 1 heading

## A level 2 heading

### A level 3 heading
```

Simple text formatting

Simple pieces of text formatting can be added by surrounding individual words or phrases with special symbols. You can use either stars (*) or underscores (_) as the special symbol. They are interchangeable but you need to use the same symbol at the start and end of the region you want to highlight.

- One star (or underscore) around a phrase makes it appear in *italics*
- Two stars (or underscores) around a phrase makes it appear in **bold**
- Three stars (or underscores) around a phrase makes it appear in ***bold italics***

```
You can make things *appear in italics* or be __bold__ or even ***do both!***
```

Lists

Having nicely rendered lists can be a useful thing. There are two kinds – ordered (which have numbered) or unordered (with bullet points).

Ordered lists are made by putting a number and a dot at the start of the line. It won't work unless you have the dot as well.

```
Things to do
------------

1. Make a list
```

```
2. Check it twice

3. Find out who's naughty or nice
```

Unordered lists just use stars at the start of the line.

```
How to develop
--------------
* Write code

* Compile

* Debug

* Repeat
```

For multi level lists you can just use tab characters at the start of the line to indent the sub-lists

```
* Write code
  * R
  * Python
  * Java
```

Tables

Most of the time the tables you use are going to be generated by your R code and not put into the markdown, but occasionally it's useful to be able to put small tables into the text portion of your document as well. As with the other markdown additions tables are designed to look nice when printed out as plain text.

The main bits of formatting are:

- Columns are separated by the pipe symbol (|)
- Headers are indicated by a row of minus symbols under them – just like a level 2 heading
- The header underline symbols can have a colon at the left or right hand or both ends to indicate the justification for the column.

```
| Name     | Quest                       | Success      |
| :------- | :-------------------------: | ------------: |
| Simon    | To teach R                  | Sometimes    |
| Emma     | To teach the world to sing | Always       |
| Libby    | To pass her GCSEs           | Unknown      |
```

In this example the Name column is left justified. This is the default so we could have omitted the colon for this column. Quest is centered and Success is right-justified.

We could do the same thing in a more compact syntax as shown below, but since the whole point of markdown is that it is somewhat readable and attractive as plain text this would mostly defeat its purpose.

```
Name|Quest|Success|
:----|:---:|---:|
Simon|To teach R|Sometimes|
Emma|To teach the world to sing|Always|
```

```
Libby|To pass her GCSEs|Unknown|
```

Fixed width text and Quotes

All R code you write in your notebook will automatically be formatted nicely for you, but sometimes you may wish to put other pieces of non-functional code into the markdown, for example to show how some of the data you're using was created.

To put in a fixed width code block you need to surround the text with triple backquotes (normally on the key to the left of '1'). Something like this.

```
The data was generated using ```grep gene bigfile.txt```
```

Another thing which is occasionally useful is to be able to quote parts of your text which will indent them and put them with a shaded background. Quoted blocks can be created by putting a ">" at the start of the line.

As Confucius said:

```
> Even the longest journey
> begins with a single step
```

Escaping

One final thing to mention about the markdown portion of the document is that sometimes you want to include some text in your document which is also a valid piece of markdown text. For example you might want to write:

```
volume = width*depth*height
```

..but the `*depth*` will be interpreted as being italic markdown so what you'll get is:

volume = width*depth*height

Since markdown characters are only special if they are immediately proceeded by text then you could just do

```
volume = width * depth * height
```

..which will be fine because of the extra spaces, but if you wanted the text to be continuous then you'll need to 'escape' it.

For any markdown formatting character that you want to be interpreted literally you just need to put a backslash in front of the relevant character. The corrected version would be:

```
volume = width\*depth\*height
```

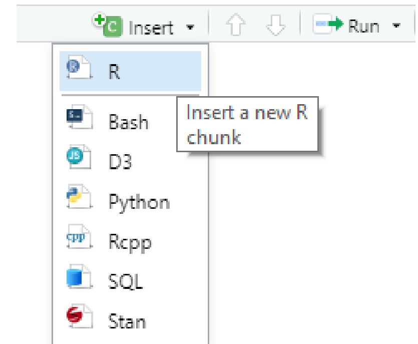
R Code

Obviously you will want to include some R code into your R Notebook and you can do this by inserting code chunks into the markdown.

Code chunks are just specially formatted pieces of code which RStudio will recognise and will allow you to run. The output from the blocks of code will be inserted directly below the code in the document, so that in the end you will build up a document with commentary, code and output all brought together.

Inserting a new code chunk

Code chunks are denoted by two special pieces of markdown, one at the start and another at the end. They look like this.




````
```{r}
# Your R code goes here
```
````

It's important that both pieces of markdown start at the beginning of the line, and that there is nothing after them. You can type the markdown in manually if you like, but generally it's easier to use the insert menu at the top of the notebook editor.

Writing R code

Once you have the markdown for your code chunk in place then you can write your R code in between the two template lines. It's important that you don't remove or alter the template lines otherwise RStudio will get confused about where your code starts and ends.

In general the code you write is exactly the same as if you were writing R into an R script file, or even just writing directly into the console. All of the code chunks are part of the same document so the code chunks are effectively concatenated together for execution. All variables created in earlier blocks are visible in the later blocks. There are only a couple of minor differences between the code you'd write in a script and the code you'd write in a Notebook chunk and we'll discuss those in a minute.

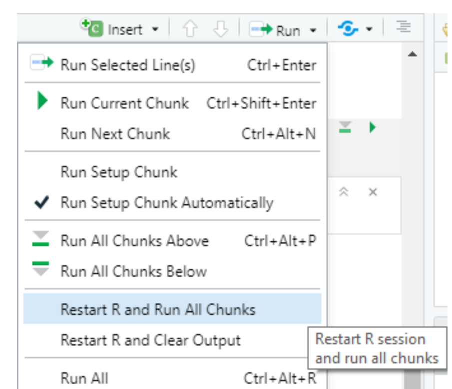
Running R code

There are a few ways to run the code in your R notebook.

In the same way as you do in an R script you can run a single statement by putting your cursor on the line and pressing Control+Return. You can also select multiple lines and do the same thing to run whatever you selected.

In general though, the most usual way to run code in a notebook is to run the whole of one chunk. You can do this by putting your cursor into the chunk you want to run and pressing Control + Shift + Return. Alternatively over every chunk there is a little green "play" arrow on the top right which you can press to run all of the code in the chunk below.

Nothing in a Notebook forces you to run the chunks in a linear order, you can go back and rerun earlier chunks at any point, but be aware that you might not get the same result as running your code in a linear fashion from the top of the document. If you have made more substantial changes to your code then you can refresh the whole document by going to the run menu at the top and selecting "Restart R and Run All Chunks". There are other options for running all or part of the document in that menu which may prove useful.



When you run some code which produces an output, the output from the code is embedded in the document immediately below that chunk. The results are then cached so that even if you close and reopen the document you will be able to see the results from the last time you ran it. Please note though that it's only the results which are cached – the actual data in your environment is not automatically restored so you won't be able to run later chunks without re-running the whole document from the top.

Breaking code into chunks

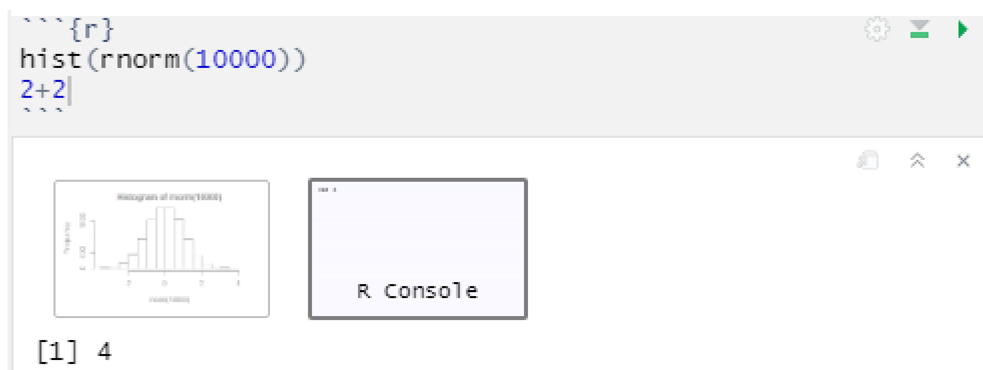
In theory you could just have one enormous code chunk, but practically it's better to keep your chunks fairly short. There are both conceptual and practical reasons for this.

From a conceptual view the advantage of having a notebook is that you can interleave commentary and code so that you have context and explanation for what you're doing. This is useful for others who will read your code, but it's also dead useful for you when you come back to the same code a week later. Having short explanations with each operation to say what you're aiming to do and what you make of the output is the ideal structure to use for maximum clarity.

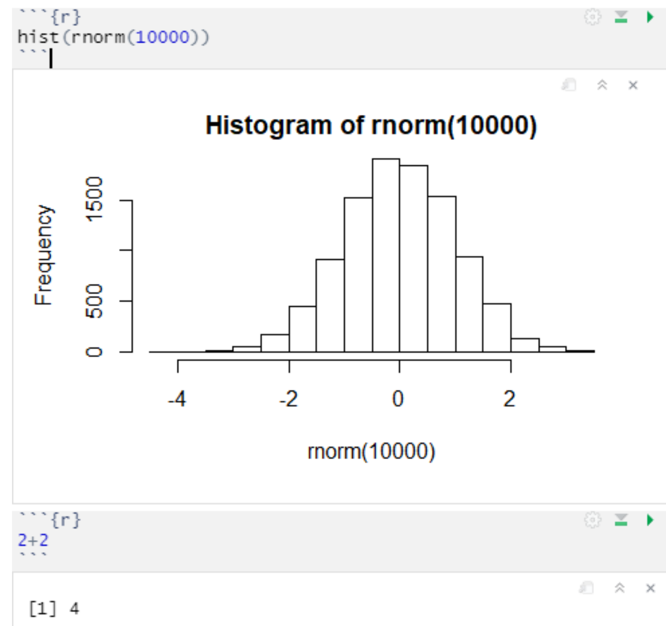
There are also some practical reasons to keep your code chunks short. Firstly, the Control+Shift+Return shortcut is the quickest way to run your code, and it runs the whole of the current chunk. If you keep your chunks short then you limit the amount of code which gets run within the chunk so development time is quicker.

Also, within the R notebook itself there is a limitation where there is only one output area for each chunk of code, which means that if your chunk generates a single piece of data or graph then you can immediately see it. If your code generates two or more outputs then they will be put into small sub-windows which you will have to click through to see all of the output. This limitation is only present in the interactive Notebook, once you render it into a document the full outputs will be created, but it's still a pain having a single chunk generate more than one output.

For example if I put two or more outputs together you can see that I get a sub-optimal presentation of the results, with only the last output from the chunk being immediately visible:



If I split these into different chunks though then all of the output is visible.



Naming chunks

We'll look at other options you can apply to your chunks a little later, but one thing which is really useful to change is to give your chunks a short name to explain what each of them is doing. This will then appear in the summary menu, alongside the header names so that it's easy to navigate around in your document.

To set a name for your chunk simply put it in the brackets after the "r"

```
```{r this is the name}
```
```

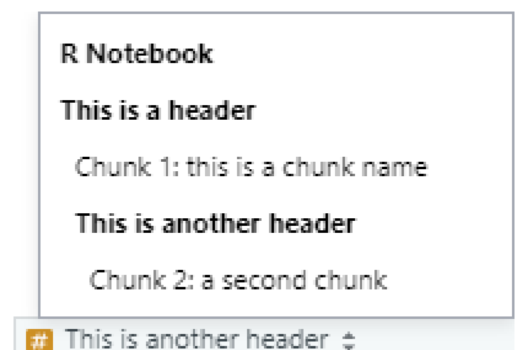
That will then cause the name to show up in the summary. So if you have something like this:

```
This is a header
================

```{r this is a chunk name}
```

This is another header

```{r a second chunk}
```
```



It will appear like this, making it easy to quickly jump to the part of the document you want to look at.

Tweaking Code Chunks

Most of the time you can just write your R code within Notebook chunks exactly the same as you would in a conventional script, sometimes it makes sense to make some adjustments to improve the presentation of the output, or to improve the readability of the code. Below we'll go through some of the things you will commonly want to change.

Differences to Core R code

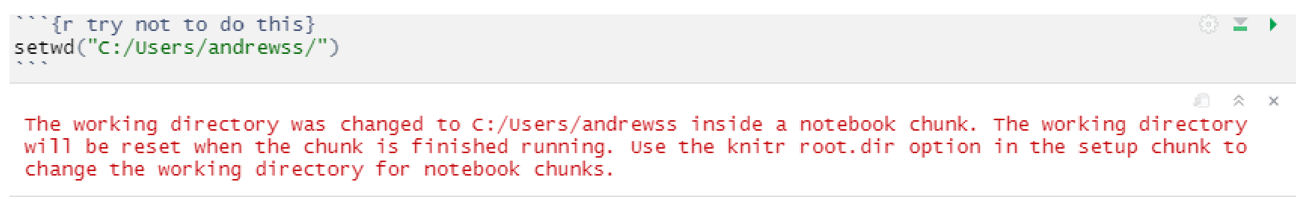
There are a couple of ways in which code within a notebook chunk behaves differently to the same code in a conventional script, and these will require you to operate in a slightly different manner.

Handling of the working directory

One of the biggest differences between notebook code and conventional code is the way that the working directory is set. In a normal R script your working directory gets some default value (eg your home directory or Documents folder) when you start an R session and you would use the `setwd` function to change this to the directory which holds your data.

Notebooks are really designed with the intention that you will put the data the notebook uses in the same directory as the notebook itself. This makes it easier to move the analysis around because you just copy the whole folder. As such, in a notebook the working directory is always set to the directory the notebook is saved into. This can catch you out when you first start writing the notebook because the working directory won't be correctly set until you have saved your notebook into its final location. It's therefore a good idea to save your notebook as soon as you create it so that the working directory is set correctly.

If you want to set your working directory to a location other than the directory the notebook is saved in then you can do this using the normal `setwd()` function, however you will get a warning generated about doing this. You should also be aware that a changed working directory only lasts for the remainder of the chunk in which you change it, so if you want to read files from a different location at several different points in your notebook then you'll need to repeatedly reset the working directory.



You can force the working directory to be different for the whole document but it's a bit of a faff. You'd need to insert a chunk at the top of your document like this one, but honestly don't do that – just put the data where the notebook is.

````
```{r setwd, cache = FALSE, include=FALSE}
require("knitr")
opts_knit$set(root.dir = "c:/Users/andrewss")
```
````

Drawing multi-layered plots

The other major difference you need to know about when writing code in notebooks is the handling of graphics. In a normal script there is a single default graphics device and all plots will be sent there unless they are explicitly saved elsewhere (ie if you send them to a file). In a notebook there is a separate graphics device for each code chunk, which is how you can have many figures and graphs inserted into the same document. However the

behaviour of the graphics devices in a notebook is slightly different to that of a conventional device – specifically the graphics devices in a notebook can't be updated once they are drawn. All of the drawing has to happen in the same operation, and must occur within a single chunk of the document.

For ggplot figures this isn't really an issue – the way that ggplot works you build all of your layers within a variable and then you use a single print statement to render that to the graphics device. There is no operation which updates an existing graph.

Core plotting routines though operate by adding graphical layers to the existing plot (the so called painters model), and this breaks if you try to run the functions individually.

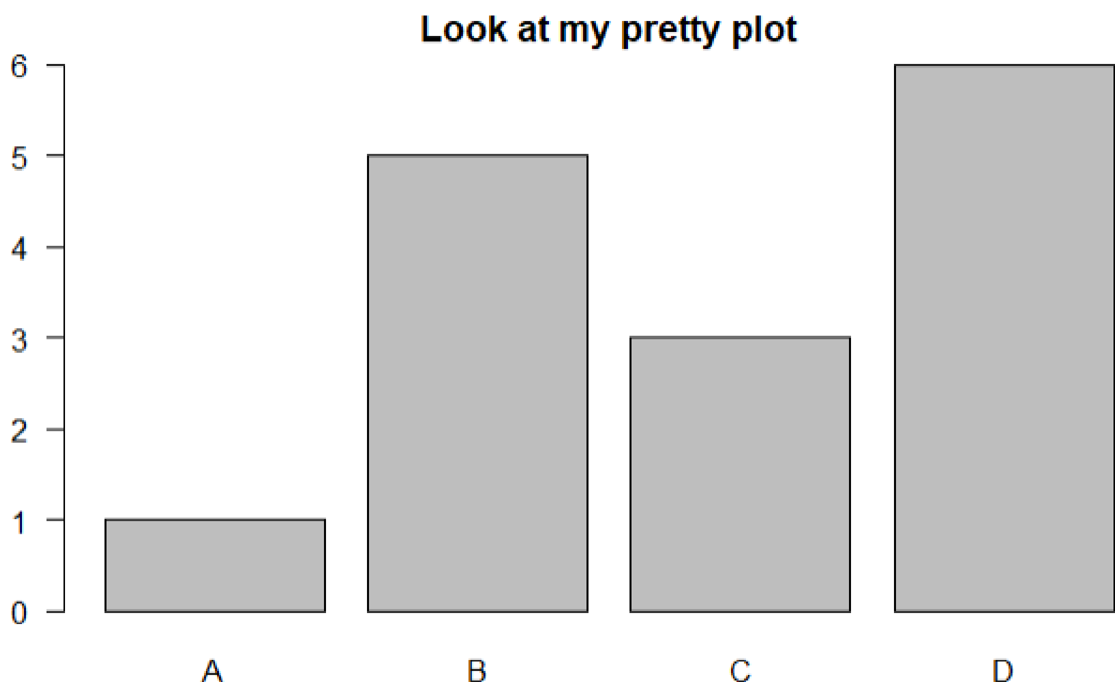
Let's consider a simple example. The code below should produce a barplot with a title

```
barplot(c(1,5,3,6),names.arg=c("A","B","C","D"),las=1)
title("Look at my pretty plot")
```

If you put this code into a single chunk, but you use Control+Enter to run the two lines separately the first line will draw the plot (without the title), and when you run the second line you'll get:

```
Error in title("Look at my pretty plot") :
  plot.new has not been called yet
```

However if you use Shift + Control + Return to run the whole chunk at once then the plot will draw correctly.



Adjusting graphical output

We said before that each code chunk gets an associated graphics output device, and this device gets some default properties. Specifically it is a PNG output device which has a size where the width is the width of the notebook document when the plot is drawn and the height is 1.6 times smaller than the width.

Often we want to draw plots which are a different size or shape, and we may wish to change the format for the plots in the finally rendered document.

Changing the size of the graphics pane

There are two places where you can change the size of graphics output. You can set values in the header to change the output sizes for all of the chunks in the document. This won't affect the way the display of the figures within the interactive document, but will change the output size when the document is rendered.

```
---
title: "R Notebook"
output:
  html_document:
    df_print: paged
    fig_width: 10
    fig_height: 10
---
```

A more flexible way to change sizes is to set sizes within individual code chunks. You can add options to the `{r}` header at the start of the chunk.

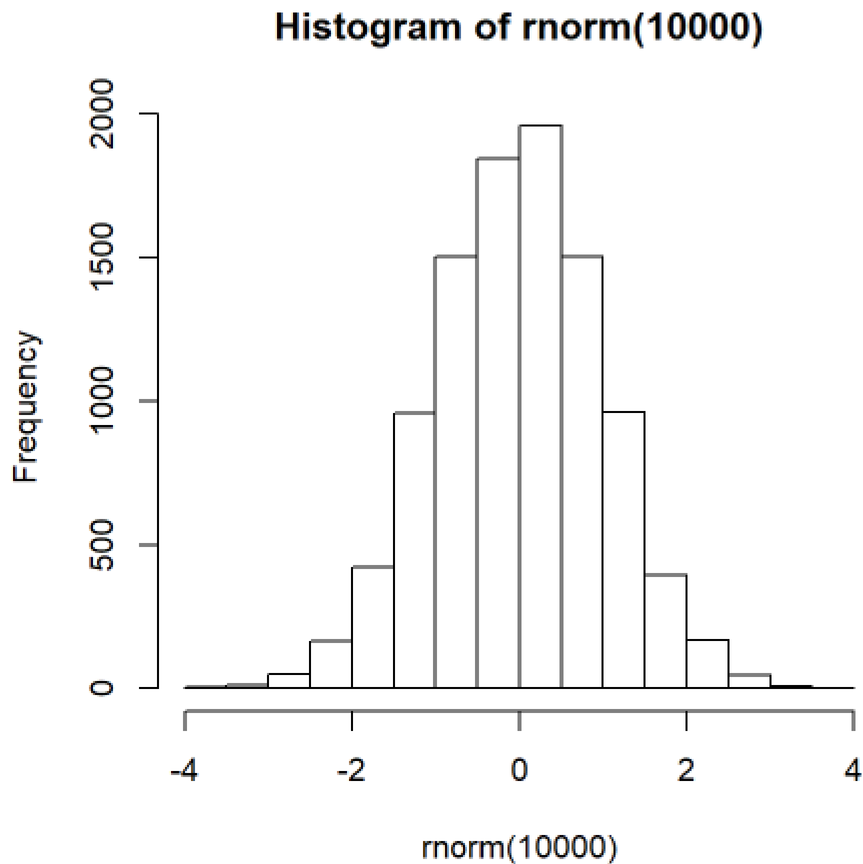
````
```{r change sizes, fig.height=5, fig.width=5, fig.align="center"}
hist(rnorm(10000))
```
````

In this instance the heights and widths are both set in inches. The plot will still be scaled so that it stays within the bounds of the interactive document so it will never fall off the right hand edge of the document but the whole thing will be scaled to fit. The correct size will be produced when the document is rendered.

The `fig.align` parameter sets where the figure is positioned on the page when it doesn't take up the whole width of the document. This setting won't affect the display in the interactive notebook, but it will take effect when the document is rendered.

The output from the code above will be a square shaped output area with the figure in the middle.

```
hist(rnorm(10000))
```



Changing the format of graphical output

By default all figures are rendered as PNG images in the rendered document. Sometimes it can be useful to change this – I often change simple figures to use SVG format so that they can be extracted from the rendered document and used as the basis for publication quality figures.

The output device type is another option which can be set in the chunk header by setting the dev parameter. To produce SVG output from a chunk you'd add dev="svg".

````
```{r multi layer plot}
barplot(c(1,5,3,6),names.arg=c("A","B","C","D"),las=1)
title("Look at my pretty plot")
```
````

````
```{r change sizes, dev="svg",fig.height=5, fig.width=5, fig.align="center"}
hist(rnorm(10000))
```
````

In the code above the rendered document would have a PNG image for the first chunk, and an SVG for the second one.

Adding a figure legend

If you want to put a short figure legend underneath the graphical output for a chunk then you can use the `fig.cap` chunk option.

```
```{r fig.height=3, fig.width=5, fig.cap="Figure3: A nice histogram"}
hist(rnorm(10000))
```
```

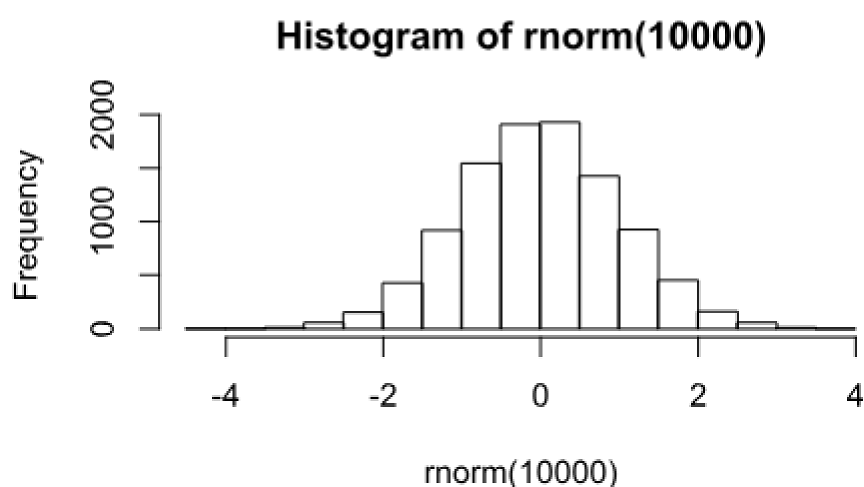


Figure3: A nice histogram

Supressing messages or warnings

A notebook will record all of the output from a chunk of code, but sometimes you want to supress some of the output to keep your rendered document a bit cleaner. There are a few options for doing this.

Supressing output in the chunk header

The simplest way to suppress output is to add more options to the chunk header to say what you do and do not want to see. Options you have are to supress the following types of output.

- Normal progress messages can be supressed with `message=FALSE`
- Warnings can be supressed with `warnings = FALSE`
- Errors will cause your notebook to stop processing. You can make it continue with `error=TRUE`. The error message
- Code – it's not a good idea to hide code in the rendered document since it won't be a complete record of your analysis but there are options which can do this.
    - You can stop the code from the chunk from appearing in your rendered document with `echo=FALSE`, the output will still show up
    - You can stop the code and output from a chunk from appearing in your rendered document with `include=FALSE` you can still use this code (eg a function definition) elsewhere in the document.

Here are some examples:

```
```{r message=FALSE}
library(tidyverse) # Supresses all of the library loading and dependency messages
```

```{r warning=FALSE}
warning("This is a warning") # Doesn't show anything
```

```{r error=TRUE}
stop("Die notebook, die")
print("Is there an afterlife?") # This will print because of the error=TRUE
```
```

Suppressing output in your code

The options above will affect all of the code in a chunk, but sometimes you want to suppress content from individual statements. There are a couple of options to do this.

`suppressMessages([code goes here])` has the same effect as message=FALSE in the header

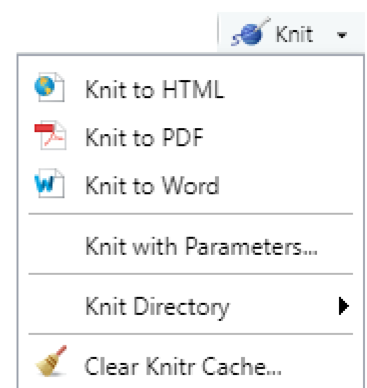
`suppressWarnings([code goes here])` has the same effect as warnings=FALSE in the header

`invisible([code goes here])` supresses the output from a function from printing. It's particularly useful if you have a function which both draws a figure and produces some output and you don't want the returned output to print.

Rendering your notebook

After you've finished writing your notebook you can then render this into a finished report. Rendering achieves two purposes – firstly it generates a nicely presented document which you can use to share the results of your analysis, but just as importantly it shows that the document you've written can be run from top to bottom in a clean environment to regenerate all of the output you expect. Within a notebook it's possible to run chunks out of order, or do other manipulations on the data which haven't been captured correctly, so knowing that your document renders is a great sanity check on your analysis.

Rendering formats



You can render your document to a number of different formats. We saw earlier that you can set and customise the format for your document in the header, but you can also choose an output format interactively from the menu above the document. This will then put the appropriate entry into the document header. You can opt to render your notebook to multiple formats at the same time if you wish.

In general I've found that rendering to HTML is the most convenient choice. Although this is an HTML document, all of the images in it are embedded rather than linked so it's a single file which you can pass along and it works well in both desktop and web based records systems.

Running a render

Once you initiate a render of your document you will see output start to stream past in the Console (it might be hidden at the bottom of the notebook but you can expand it). You will see the results of rendering each of the chunks and finally the generation of the finished documents.

If something fails you will get a stack trace to show where it went wrong so you can work out how to fix it. Generally there's no reason to get an R failure in a render which you wouldn't have seen by using the "Restart R and run all chunks" option in the Run menu. You can sometimes get failures which will only be apparent in the render. These normally have to do with bits of the code chunk templates going missing (eg the final ``` being deleted or tacked onto the end of the line before), or from malformed options in the code chunk header (normally missing commas between different options).

Changing the rendered document's appearance

We saw when we looked at the header options that you can make some changes to the appearance of a notebook by altering the header. The main things you're likely to want to do are to add a table of contents, and maybe to have this float on the left of the page where it's always visible.

In addition to these functional changes you can also make more cosmetic alterations to your document. Specifically R comes with a set of Themes for notebooks and you can quickly change the appearance of your document by simply selecting an alternate theme.

Setting a theme and highlighting colour scheme is done as shown below:

```
---
title: "Themes"
output:
  html_document:
    df_print: paged
    toc: true
    toc_float: true
    theme: yeti
    highlight: kate
---
```

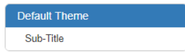
As with the table of contents, themes are also set in the document header. There are two type of theme you can change:

1. The design theme for the whole document – controls heading colours and fonts

2. The syntax highlighting theme – controls how the code chunks are coloured

Document Themes

Document themes are taken from a project called Bootswatch (bootswatch.com) but only some of them are available in R. The themes you can use and their appearance are shown below.



Themes

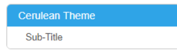
Default Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

Cerulean Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.

Themes

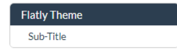
Journal Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

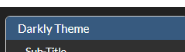
Flatly Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

Darkly Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.

Themes

Readable Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.

Themes

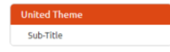
Spacelab Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

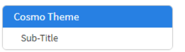
United Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

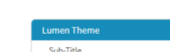
Cosmo Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

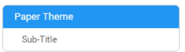
Lumen Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

Paper Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.

Themes

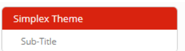
Sandston Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

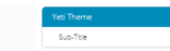
Simplex Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.



Themes

Yeti Theme

This is a small document to show the effect of changing the themes.

Sub-Title

- Themes can change
- The overall appearance

of your document in a quick and easy fashion.

Highlight themes

These are your options for doing syntax highlighting.

```
# Tango

library(tidyverse)

starwars %>%
  filter(height>150) %>%
  arrange(desc(birth_year)) %>%
  filter(gender=="male")
```

```
# Pygments

library(tidyverse)

starwars %>%
  filter(height>150) %>%
  arrange(desc(birth_year)) %>%
  filter(gender=="male")
```

```
# Kate

library(tidyverse)

starwars %>%
  filter(height>150) %>%
  arrange(desc(birth_year)) %>%
  filter(gender=="male")
```

```
# Monochrome

library(tidyverse)

starwars %>%
  filter(height>150) %>%
  arrange(desc(birth_year)) %>%
  filter(gender=="male")
```

```
# Espresso

library(tidyverse)

starwars %>%
  filter(height>150) %>%
  arrange(desc(birth_year)) %>%
  filter(gender=="male")
```

```
# Zenburn

library(tidyverse)

starwars %>%
  filter(height>150) %>%
  arrange(desc(birth_year)) %>%
  filter(gender=="male")
```

```
# Haddock

library(tidyverse)

starwars %>%
  filter(height>150) %>%
  arrange(desc(birth_year)) %>%
  filter(gender=="male")
```

```
# Textmate


library(tidyverse)


starwars %>%

  filter(height>150) %>%

  arrange(desc(birth_year)) %>%

  filter(gender=="male")
```