© Thomas Mailund 2022

T. Mailund, *Beginning Data Science in R 4*

https://doi.org/10.1007/978-1-4842-8155-0_2

2. Reproducible Analysis

Thomas Mailund[1]

(1) Aarhus, Denmark

The typical data analysis workflow looks like this: you collect your data, and you put it in a file or spreadsheet or database. Then you run some analyses, written in various scripts, perhaps saving some intermediate results along the way or maybe always working on the raw data. You create some plots or tables of relevant summaries of the data, and then you go and write a report about the results in a text editor or word processor. This is the typical workflow in many organizations and in many research groups. Most people doing data analysis do variations thereof. But it is also a workflow that has many potential problems.

There is a separation between the analysis scripts and the data, and there is a separation between the analysis and the documentation of the analysis.

If all analyses are done on the raw data, then issue number one is not a major problem. But it is common to have scripts for different parts of the analysis, with one script saving intermediate results to files that are then read by the next script. The scripts describe a workflow of data analysis, and to reproduce an analysis, you have to run all the scripts in the right order. Often enough, this correct order is only described in a text file or even worse only in the head of the data scientist who wrote the workflow. And it gets worse; it won't stay there for long and is likely to be lost before it is needed again.

Ideally, you would always want to have your analysis scripts written in a way where you can rerun any part of your workflow, completely automatically, at any time.

For issue number two, the problem is that even if the workflow is automated and easy to run again, the documentation quickly drifts away from the actual analysis scripts. If you change the scripts, you won't necessarily remember to update the documentation. You probably don't forget to update figures and tables and such, but not necessarily the documentation of the exact analysis run—options to functions and filtering choices and such. If the documentation drifts far enough from the actual analysis, it becomes completely useless. You can trust automated scripts to represent the real data analysis at any time—that is the benefit of having automated analysis workflows in the first place—but the documentation can easily end up being pure fiction.

What you want is a way to have dynamic documentation. Reports that describe the analysis workflow in a form that can be understood both by machines and humans. Machines use the report as an automated workflow that can redo the analysis at any time. We humans use it as documentation that always accurately describes the analysis workflow that we run.

Literate Programming and Integration of Workflow and Documentation

One way to achieve the goal of having automated workflows and documentation that is always up to date is something called "literate programming." Literate programming is an approach to software development, proposed by Stanford computer scientist Donald Knuth, which never became popular for programming, possibly because most programmers do not like to write documentation. But it has made a comeback in data science, where tools such as Jupyter Notebooks[1] and R Markdown (that we will explore later) are major components in many data scientists' daily work.

The idea in literate programming is that the documentation of a program—in the sense of the documentation of how the program works and how algorithms and data structures in the program work—is written together with the code implementing the program. Tools such as Javadoc[2] and Roxygen[3] do something similar. They have documentation of classes and methods written together with the code in the form of comments. Literate programming differs slightly from this. With Javadoc and Roxygen, the code is the primary document, and the documentation is comments added to it. With literate programming, the documentation is the primary text for humans to read, and the code is part of this documentation, included where it falls naturally to have it. The computer code is extracted automatically from this document when the program runs.

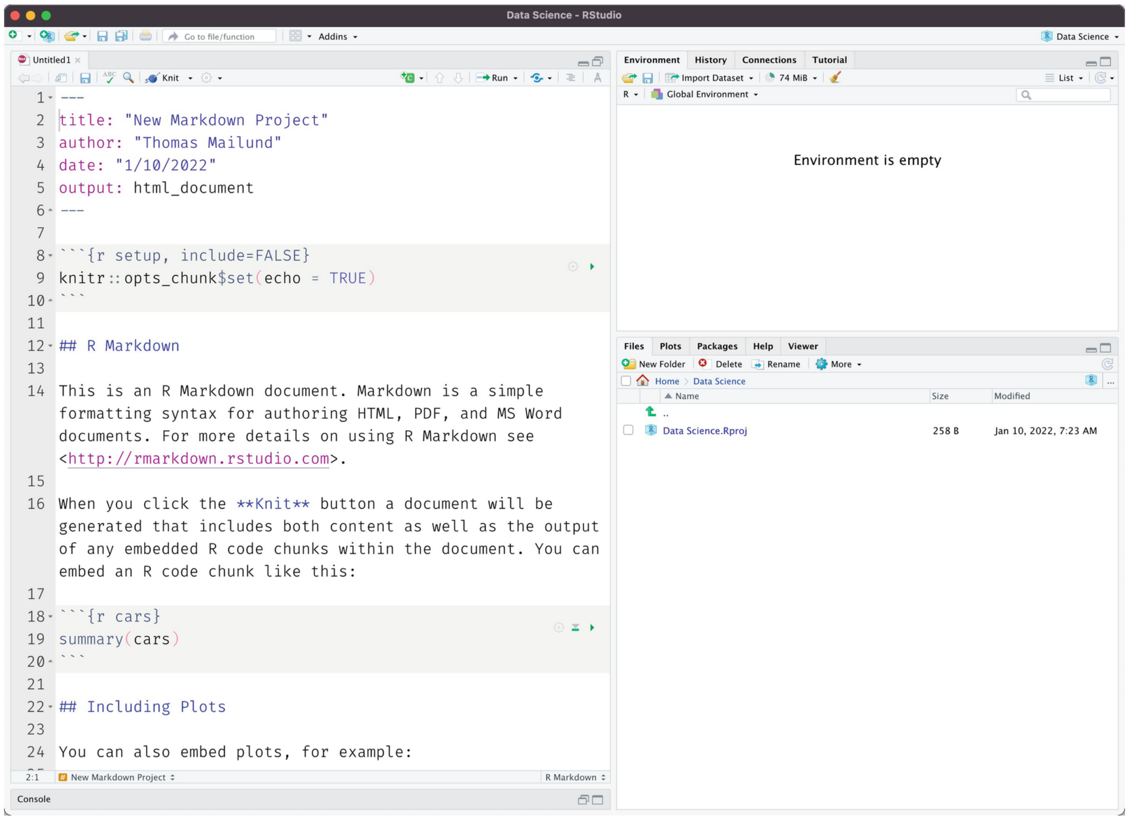
Literate programming never became a huge success for writing programs, but for doing data science, it is having a comeback. The result of a data analysis project is typically a report describing models and analysis results, and it is natural to think of this document as the primary product. So the documentation is already the main focus. The only thing needed to use literate programming is a way of putting the analysis code inside the documentation report.

Many programming languages have support for this. Mathematica[4] has always had notebooks where you could write code together with documentation. Jupyter,[5] the descendant of iPython Notebook, lets you write notebooks with documentation and graphics interspersed with executable code. And in R there are several ways of writing documents that are used both as automated analysis scripts and for generating reports. The most popular of these approaches is R Markdown (for writing these documents) and `knitr` (for running the analysis and generating the reports), but R Notebooks, a variant of R Markdown, is also gaining popularity.

Creating an R Markdown/knitr Document in RStudio

To create a new R Markdown document, go to the File menu, pick New File and then R Markdown.... Now RStudio will bring up a dialog where you can decide which kind of document you want to make and add some information, such as title and author name. It doesn't matter so much what you do here, you can change it later, but try making an HTML document.

The result is a new file with some boilerplate text in it; see Figure 2-1. At the top of the file, between two lines containing just "---" is some meta-information for the document, and after the second "---" is the text proper. It consists of a mix of text, formatted in the Markdown language, and R code.



~~~
---
title: "New Markdown Project"
author: "Thomas Mailund"
date: "1/10/2022"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
~~~

R Markdown

This is an R Markdown document. Markdown is a simple
formatting syntax for authoring HTML, PDF, and MS Word
documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be
generated that includes both content as well as the output
of any embedded R code chunks within the document. You can
embed an R code chunk like this:

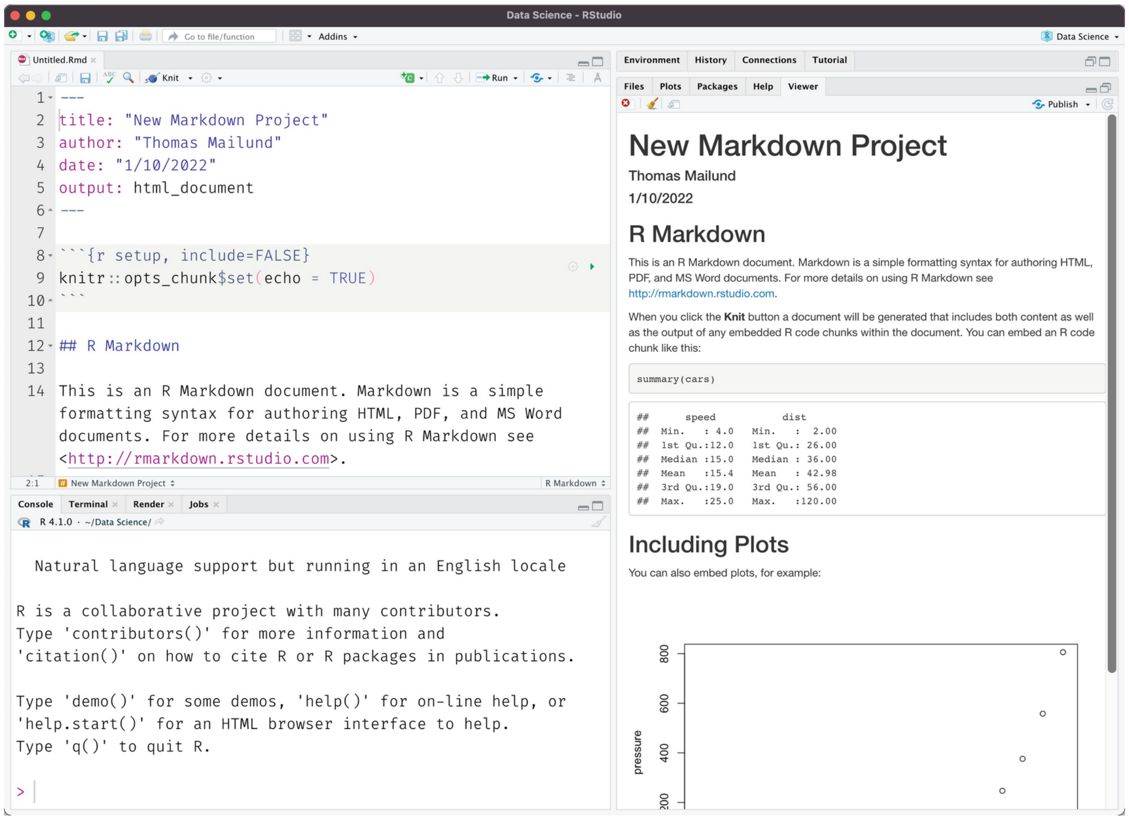
```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:
```

Figure 2-1 New R Markdown file

In the toolbar above the open file, there is a menu point saying Knit. If you click it, it will translate the R Markdown into an HTML document and open it; see Figure 2-2. You will have to save the file first, though. If you click the Knit HTML button before saving, you will be asked to save the file.
```



~~~
---
title: "New Markdown Project"
author: "Thomas Mailund"
date: "1/10/2022"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

R Markdown

This is an R Markdown document. Markdown is a simple
formatting syntax for authoring HTML, PDF, and MS Word
documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.
~~~

```
  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

New Markdown Project

Thomas Mailund

1/10/2022

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      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  
```

Including Plots

You can also embed plots, for example:

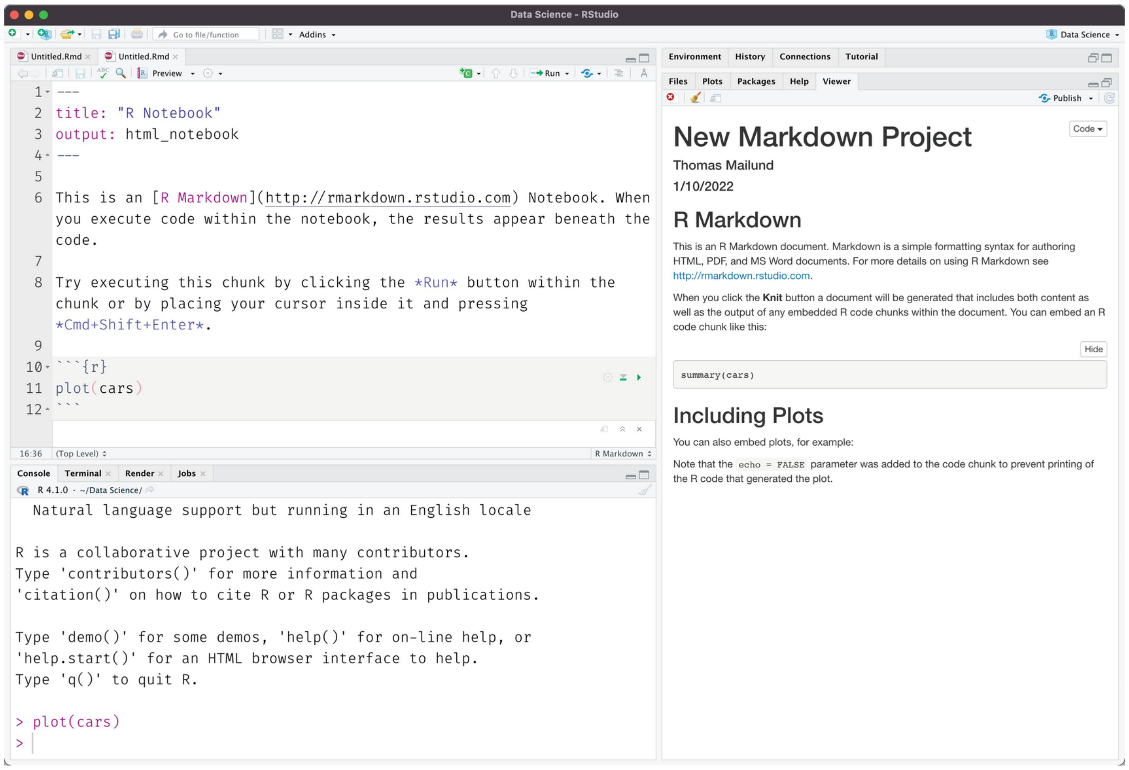
Figure 2-2 Compiled Markdown file

The newly created HTML file is also written to disk with a name taken from the name you gave the R Markdown file. The R Markdown file will have suffix `.Rmd`, and the HTML file will have the same prefix but suffix `.html`.

If you click the down-pointing arrow next to Knit, you get some additional options. You can ask to see the HTML document in the pane to the right in RStudio instead of in a new window. Having the document in a panel instead of a separate window can be convenient if you are on a laptop and do not have a lot of screen space. You can also generate a file or a Word file instead of an HTML file.

If you decide to produce a file in a different output format, RStudio will remember this. It will update the “output:” field in the metadata to reflect this. If you want, you can also change that line in your document and make the selection that way. Try it out.

If you had chosen File, New File, R Notebook instead of an R Markdown file, you would have gotten a very similar file; see Figure 2-3. The Knitr button is gone, and instead you have a Preview button. If you click it, you get the document shown on the right in Figure 2-3. The difference between the two types of files is tiny, to the point where it doesn't exist. The Notebook format is just a different output option, and both _are_ R Markdown files. If you had changed the line "output: html_document" in the first file to "output: html_notebook", you would get a notebook instead. Try it and see what happens.



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

This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When
you execute code within the notebook, the results appear beneath the
code.

Try executing this chunk by clicking the *Run* button within the
chunk or by placing your cursor inside it and pressing
*Cmd+Shift+Enter*.

```{r}
plot(cars)
```
~~~

```
  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> plot(cars)
> 
```

New Markdown Project

Thomas Mailund

1/10/2022

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

Including Plots

You can also embed plots, for example:

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

Figure 2-3 R Notebook

The R Markdown file is intended for creating reports, while the R Notebook format is more intended for interactive use. Knitting a large document can be slow, because all the analysis in your document will be run from scratch, but with the notebooks, you see the document as it is at any moment, without rerunning any analysis when you preview. Notebooks are thus faster, but they can be a little more dangerous to work with, if you evaluate code out of order (see the following). You can, however, always work with a Notebook while you do your analysis, and then change the "output: ..." format later, to generate a report from scratch. Because the formats are so similar, I will not distinguish between them in the following, and I will refer to the files as R Markdown files as that is the input format for both of them (as also apparent from their file suffix ".Rmd").

The actual steps in creating a document involve two tools and three languages, but it is all integrated so you typically will not notice. There is the R code embedded in the document. The R code is first processed by the `knitr` package that evaluates it and handles the results such as data and plots according to options you give it. The result is a Markdown document (notice no R). This Markdown document is then processed by the tool `pandoc` which is responsible for generating the output file. For this, it uses the metadata in the header, which is written in a language called YAML, and the actual formatting, written in the Markdown language.

You will usually never have to worry about `pandoc` working as the back end of document processing. If you just write R Markdown documents, then RStudio will let you compile them into different types of output documents. But because the pipeline goes from R Markdown via `knitr` to Markdown and then via `pandoc` to the various output formats, you do have access to a very powerful tool for creating documents. I have written this book in R Markdown where each chapter is a separate document that I can run through `knitr` independently. I then have `pandoc` with some options take the resulting Markdown documents, combining them, and produce both output and Epub output. With `pandoc`, it is possible to use different templates for setting up the formatting, and having it depend on the output document you create by using different templates for different formats. It is a very powerful, but also very complicated, tool, and it is far beyond what we can cover in this book. Just know that it is there if you want to take document writing in R Markdown further than what you can readily do in RStudio.

As I mentioned, there are actually three languages involved in an R Markdown document. We will handle them in order, first the header language, which is YAML, then the text formatting language, which is Markdown, and then finally how R is embedded in a document.

The YAML Language

YAML is a language for specifying key-value data. YAML stands for the (recursive) acronym YAML Ain’t Markup Language. So yes, when I called this section the YAML language I shouldn’t have included language since the L stands for language, but I did. I stand by that choice. The acronym used to stand for Yet Another Markup Language, but since “markup language” typically refers to commands used to mark up text for either specifying formatting or for putting structured information in a text, which YAML doesn’t do, the acronym was changed. YAML is used for giving options in various forms to a computer program processing a document, not so much for marking up text, so it isn’t really a markup language.

In your R Markdown document, the YAML is used in the header, which is everything that goes between the first and the second line with three dashes. In the document you create when you make a new R Markdown file, it can look like this:

```
---
title: "New Markdown Project"
author: "Thomas Mailund"
date: "1/10/2022"
output: html_document
---
```

You usually do not have to modify this header manually. If you use the GUI, it will adjust the header for you when you change options. You do need to alter it to include bibliographies, though, which we get to later. And you can always add anything you want to the header if you need to, and it can be easier than using the GUI. But you don’t have to modify the header that often.

YAML gives you a way to specify key-value mappings. You write `key:` and then the `value` afterward. So in the preceding example, you have the key `title` referring to `New Markdown Document`, the key `author` to refer to `"Thomas Mailund"`, and so on. You don’t necessarily need to quote the values unless it has a colon in it, but you always can.

The YAML header is used both by RStudio and `pandoc`. RStudio uses the `output` key for determining which output format to translate your document into, and this choice is reflected in the Knit toolbar button—while `pandoc` uses the `title`, `author`, and `date` to put that information into the generated document.

You can have slightly more structure to the key-value specifications. If a key should refer to a list of values, you use "–", so if you have more than one author, you can use something like this:

```
---
...
author:
 - "Thomas Mailund"
 - "Tom Maygrove"
...
---
```

or you can have more key-value structure nested, so if you change the output theme (using Output Options… after clicking the tooth wheel in the toolbar next to the Knit button).

How the options are used depends on the toolchain used to format your document. The YAML header just provides specifications. Which options you have available and what they do are not part of the language.

For `pandoc`, it depends on the templates used to generate the final document (see later), so there isn't even a complete list that I can give you for `pandoc`. Anyone who writes a new template can decide on new options to use. The YAML header gives you a way to provide options to such templates, but there isn't a fixed set of keywords to use. It all depends on how tools later in the process interpret them.

The Markdown Language

The Markdown language is a markup language—the name is a pun. It was originally developed to make it easy to write web pages. HTML, the language we use to format web pages, is also a markup language but is not always easily human readable. Markdown intended to solve this by formatting text with very simple markup commands—familiar from emails back in the day before emails were also HTML documents—and then have tools for translating Markdown into HTML.

Markdown has gone far beyond just writing web pages, but it is still a very simple and intuitive language for writing human-readable text with markup commands that can then be translated into other document formats.

In Markdown, you write plain text as plain text. So the body of text is just written without any markup. You will need to write it in a text editor so the text is actually text, not a word processor where the file format usually already contains a lot of markup information that just isn't readily seen on screen. If you are writing code, you should already know about text editors. If not, just use RStudio to write R Markdown files, and you will be okay.

Markup commands are used when you want something else than just plain text. There aren't many commands to learn—the philosophy is that when writing you should focus on the text and not the formatting—so they are very quickly learned.

Formatting Text

First, there are section headers. You can have different levels of headers—think chapters, sections, subsections, etc.—and you specify them using `#` starting at the beginning of a new line:

```
# Header 1
## Header 2
### Header 3
```

For the first two, you can also use this format:

```
Header 1
========
Header 2
-------------
```

To have lists in your document, you write them as you have probably often seen them in raw text documents. A list with bullets (and not numbers) is written like this:

```
* this is a
* bullet
* list
```

and the result looks like this:

- this is a
- bullet
- list

You can have sublists just by indenting. You need to move the indented line in so there is a space between where the text starts at the outer level and where the bullet is at the next level. Otherwise, the line goes at the outer level. The output of this

```
* This is the first line
   * This is a sub-line
   * This is another sub-line
  * This actually goes to the outer level
* This is definitely at the outer level
```

is this list:

- This is the first line
  - – This is a sub-line
  - – This is another sub-line
  - – This actually goes to the outer level
- Back to the outer level

If you prefer, you can use – instead of `*` for these lists, and you can mix the two:

```
- First line
* Second line
   - nested line
```

- First line
- Second line
  - – nested line

To have numbered lists, just use numbers instead of `*` and `–`:

```
1. This is a
2. numbered
```

```
3. list
```

The result looks like this:

1. This is a
2. numbered
3. list

You don't actually need to get the numbers right, you just need to use numbers. So

```
1. This is a
3. numbered
2. list
```

would produce the same output. You will start counting at the first number, though, so

```
4. This is a
4. numbered
4. list
```

produces

4. This is a
5. numbered
6. list

To construct tables, you also use a typical text representation with vertical and horizontal lines. Vertical lines separate columns, and horizontal lines separate headers from the table body. This code

```
| First Header  | Second Header | Third Header    |
| :------------ | :-----------: | --------------: |
| First row     | Centred text  | Right justified |
| Second row    | *Some data*   | *Some data*     |
| Third row     | *Some data*   | *Some data*     |
```

will result in this table:

First Header	**Second Header**	**Third Header**
First row	Centred text	Right justified
Second row	Some data	Some data

Third row	Some data	Some data

The `:` in the line separating the header from the body determines the justification of the column. Put it on the left to get left justification, on both sides to get the text centered, and on the right to get the text right justified.

Inside text, you use markup codes to make text italic or boldface. You use either `*this*` or `_this_` to make “this” italic, while you use `**this**` or `__this__` to make “this” boldface.

Since Markdown was developed to make HTML documents, it, of course, has an easy way to insert links. You use the notation `[link text](link URL)` to put “link text” into the document as a link to “link URL.” This notation is also used to make cross-references inside a document—similar to how HTML documents have anchors and internal links—but more on that later.

To insert images into a document, you use a notation similar to the link notation; you just put a `!` before the link, so `![Image description](URL to image)` will insert the image pointed to by “URL to image” with a caption saying “Image description.” The URL here will typically be a local file, but it can be a remote file referred to via HTTP.

With long URLs, the marked-up text can be hard to read even with this simple notation, and it is possible to remove the URLs from the actual text and place it later in the document, for example, after the paragraph referring to the URL or at the very end of the document. For this, you use the notation `[link text][link tag]` and define the “link tag” as the URL you want later:

```
This is some text [with a link][1].
The link tag is defined below the paragraph.
[1]: interesting-url-of-some-sort-we-dont-want-inline
```

You can use a string here for the tag. Using numbers is easy, but for long documents, you won’t be able to remember what each number refers to:

```
This is some text [with a link][interesting].
The link tag is defined below the paragraph.
```

```
[interesting]: interesting-url-of-some-sort-we-dont-want-
inline
```

You can make block quotes in text using notation you will be familiar with from emails:

```
> This is a
> block quote
```

gives you this:

> *This is a block quote*

To put verbatim input as part of your text, you can either do it inline or as a block. In both cases, you use backticks `. Inline in the text, you use single backticks `` `foo` ``. To create a block of text, you write

    ```
    block of text
    ```

You can also just indent text with four spaces, which is how I managed to make a block of verbatim text that includes three backticks.

Markdown is used a lot by people who document programs, so there is a notation for getting code highlighted in verbatim blocks. The convention is to write the name of the programming language after the three backticks, then the program used for formatting the document will highlight the code when it can. For R code, you write `r`, so this block

    ```r
    f <- function(x) ifelse(x %% 2 == 0, x**2, x**3)
    f(2)
    ```

is formatted like this:

```r
f <- function(x) ifelse(x %% 2 == 0, x**2, x**3)
f(2)
```

The only thing this markup of blocks does is highlighting the code. It doesn’t try to evaluate the code. Evaluating code happens before the Markdown document is formatted, and we return to that shortly.

Cross-Referencing

Out of the box, there is not a lot of support for making cross-references in Markdown documents. You can make cross-references to sections but not figures or tables. There are ways of doing it with extensions to `pandoc`—I use it in this book—but out of the box from RStudio, you cannot yet.

However, with the work being done for making book-writing and lengthy reports in Bookdown,[6] that might change soon.[7]

The easiest way to reference a section is to put the name of the section in square brackets. If I write `[Cross referencing]` here, I get a link to this Cross referencing section. Of course, you don't always want the name of the section to be the text of the link, so you can also write `[this section][Cross referencing]` to get a link to the section "Cross referencing" but display the text "this section."

This approach naturally only works if all section titles are unique. If they are not, then you cannot refer to them simply by their names. Instead, you can tag them to give them a unique identifier. You do this by writing the identifier after the title of the section. To put a name after a section header, you write

```
### Cross referencing {#section-cross-ref}
```

and then you can refer to the section using `[this](#section-cross-ref)`. Here, you do need the `#` sign in the identifier—that markup is leftover from HTML where anchors use `#`.

Bibliographies

Often, you want to cite books or papers in a report. You can of course always handle citations manually, but a better approach is to have a file with the citation information and then refer to it using markup tags. To add a bibliography, you use a tag in the YAML header called `bibliography`:

```
---
...
bibliography: bibliography.bib
...
---
```

You can use several different formats here; see the R Markdown documentation[8] for a list. The suffix `.bib` is used for BibLaTeX. The format for the citation file is the same as BibTeX, and you get citation information in that format from nearly every site that will give you bibliography information.

To cite something from the bibliography, you use `[@smith04]` where `smith04` is the identifier used in the bibliography file. You can cite more than one paper inside square brackets separated by a semicolon, `[@smith04; doe99]`, and you can add text such as chapters or page numbers `[@smith04, chapter 4]`. To suppress the author name(s) in the citation, say when you mention the name already in the text, you put – before the `@` so you write `As Smith showed [-@smith04]...`. For in-text citations, similar to `\citet{}` in `natbib`, you just leave out the brackets, `@smith04 showed that...`, and you can combine that with additional citation information as `@smith04 [chapter 4] showed that...`.

To specify the citation style to use, you use the `csl` tag in the YAML header:

```
---
...
bibliography: bibliography.bib
csl: biomed-central.csl
...
---
```

Check out the list of citation styles at https://github.com/citation-style-language/styles for a large number of different formats. There should be most if not all your heart desires.

Controlling the Output (Templates/Stylesheets)

The `pandoc` tool has a powerful mechanism for formatting the documents it generates. This is achieved using stylesheets in CSS for HTML and from using templates for how to format the output for all output formats. The template mechanism lets you write an HTML or LaTeX document, say, that determines where various part of the text goes and where variables from the YAML header are used. This mechanism is far beyond what we can cover in this chapter, but I just want to mention it if you want to start writing papers using R Markdown. You can do this; you just need to have a template for formatting the document in the style a journal wants. Often, they provide LaTeX templates, and you can modify these to work with Markdown.

There isn't much support for this in RStudio, but for HTML documents, you can use the Output Options… (click the tooth wheel) to choose different output formatting.

Running R Code in Markdown Documents

The formatting so far is all Markdown (and YAML). Where it combines with R and makes it R Markdown is through `knitr`. When you format a document, the first step evaluates R code to create a Markdown document—this translates an `.Rmd` document into an `.md` document, but this intermediate document is deleted afterward unless you explicitly tell RStudio not to do it. It does that by running all the R code you want to be executed and putting it into the Markdown document.

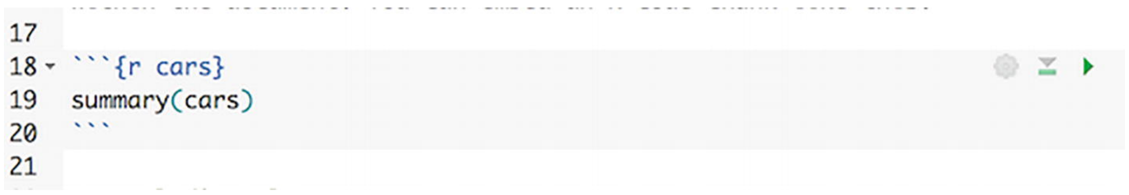
The simplest R code you can evaluate is part of a text. If you want an R expression evaluated, you use backticks but add `r` right after the first. So to evaluate `2 + 2` and put the result in your Markdown document, you write `` `r `` and then the expression `2 + 2` and get the result 4 inserted into the text. You can write any R expression there to get it evaluated. It is useful for inserting short summary statistics like means and standard deviations directly into the text and ensuring that the summaries are always up to date with the actual data you are analyzing.

For longer chunks of code, you use the block quotes, the three backticks. Instead of just writing

~~~
```r
2 + 2
```
~~~

which will only display the code (highlighted as R code), you put the `r` in curly brackets.

This will insert the code in your document but also show the result of evaluating it right after the code block. The boilerplate code you get when creating an R Markdown document in RStudio shows you examples of this (see Figure 2-4).



~~~
```{r cars}
summary(cars)
```
~~~

Figure 2-4 Code chunk in RStudio

You can name code chunks by putting a name right after `r`. You don't have to name all chunks—and if you have a lot of chunks, you probably won't bother naming all of them—but if you give them a name, they are easily located by clicking the structure button in the bar below the document (see Figure 2-5). You can also use the name to refer to chunks when caching results, which we will cover later.

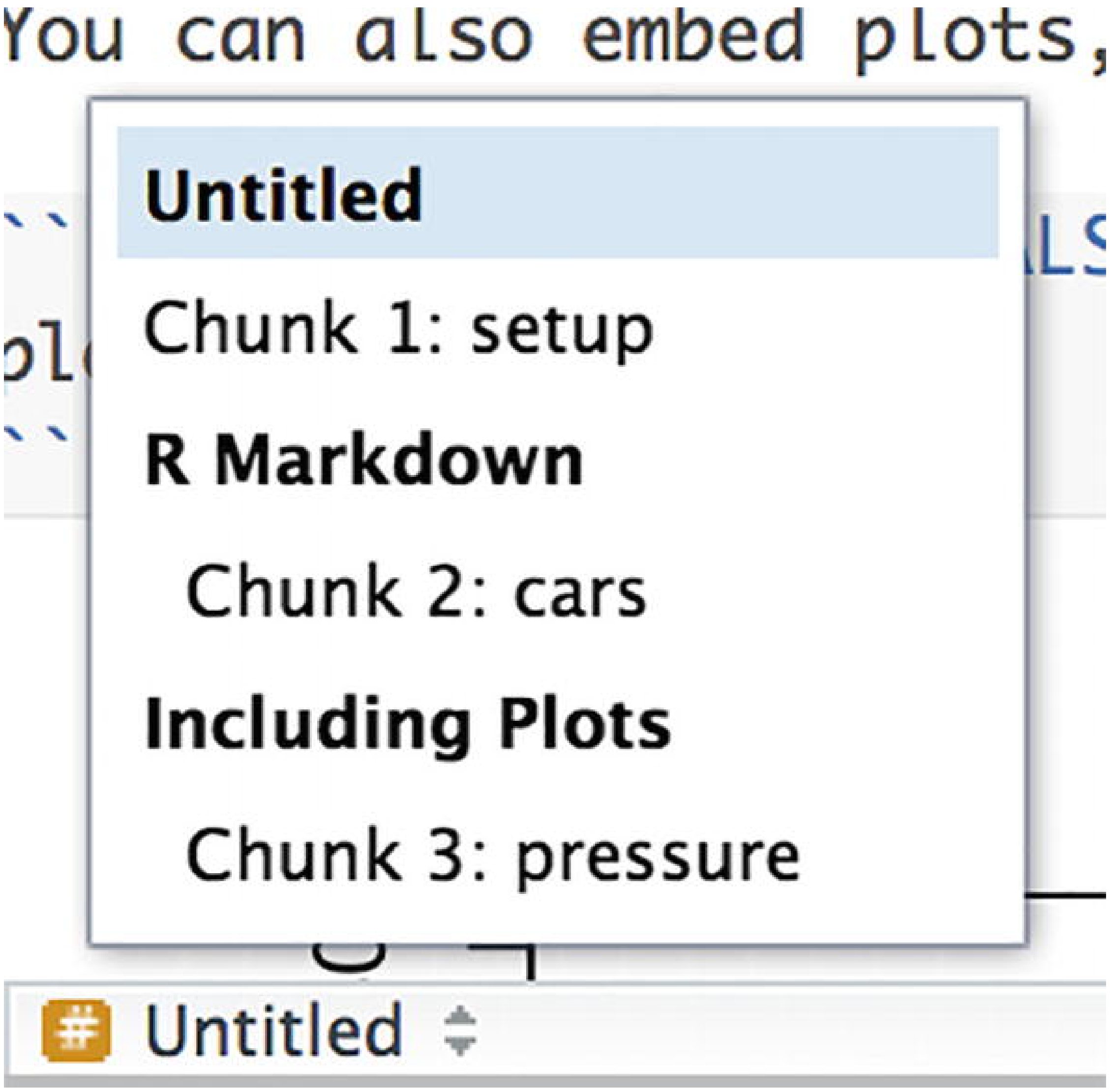
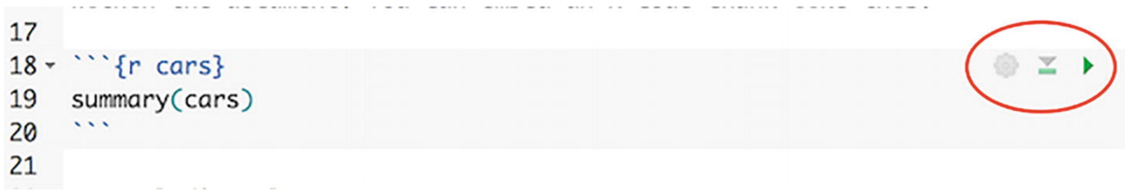


Figure 2-5 Document structure with chunk names

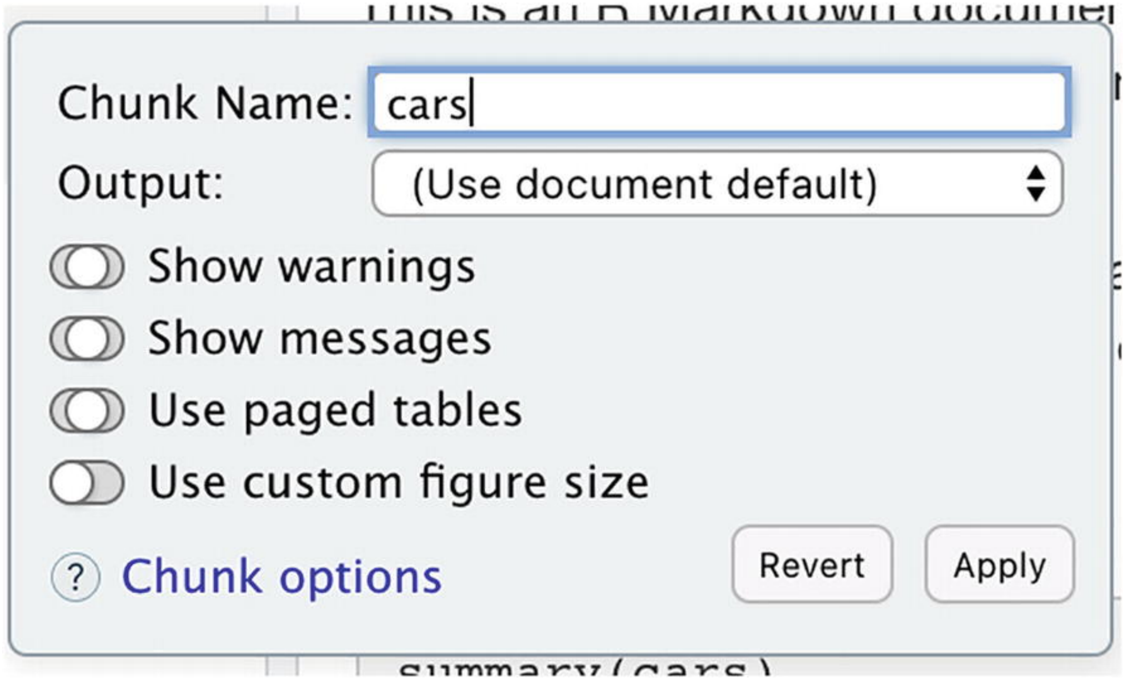
You should see a toolbar to the right on every code chunk (see Figure 2-6). The rightmost option, the "play" button, will let you evaluate the chunk. The results will be shown below the chunk unless you have disabled that option. The middle button evaluates all previous chunks down to and including the current one. This is useful when the current chunk depends on previous results. The tooth wheel lets you set options for the chunk.



~~~
```{r cars}
summary(cars)
```
~~~

Figure 2-6 Code chunk toolbar

The chunk options (see Figure 2-7) control the output you get when evaluating a code chunk. The Output drop-down selects what output the chunk should generate in the resulting document, while the Show warnings and Show messages selection buttons determine whether warnings and messages, respectively, should be included in the output. The "Use paged tables" changes how tables are displayed, splitting large tables into pages you can click through. The Use custom figure size is used to determine the size of figures you generate —but we return to these later.



Chunk Name: cars
Output: (Use document default)

Figure 2-7 Code chunk options

If you modify these options, you will see that the options are included in the top line of the chunk. You can of course also manually control the options here, and there are more options than what you can control with the dialog in the GUI. You can read the knitr documentation[9] for all the details.

The dialog will handle most of your needs, though, except for displaying tables or when we want to cache results of chunks, both of which we return to later.

Using chunks when analyzing data (without compiling documents)

Before continuing, though, I want to stress that working with data analysis in an R Markdown document is useful for more than just creating documents. I personally do all my analysis in these documents because I can combine documentation and code, regardless of whether I want to generate a report at the end. The combination of explanatory text and analysis code is just convenient to have.

The way code chunks are evaluated as separate pieces of analysis is also part of this. You can evaluate chunks individually, or all chunks down to a point, and I find that very convenient when doing an analysis. There are keyboard shortcuts for evaluating all chunks, all previous chunks, or just the current chunk (see Figure 2-8), which makes it very easy to write a bit of code for an exploratory analysis and evaluating just that piece of code. If you are familiar with Jupyter, or similar notebooks, you will recognize the workflow.

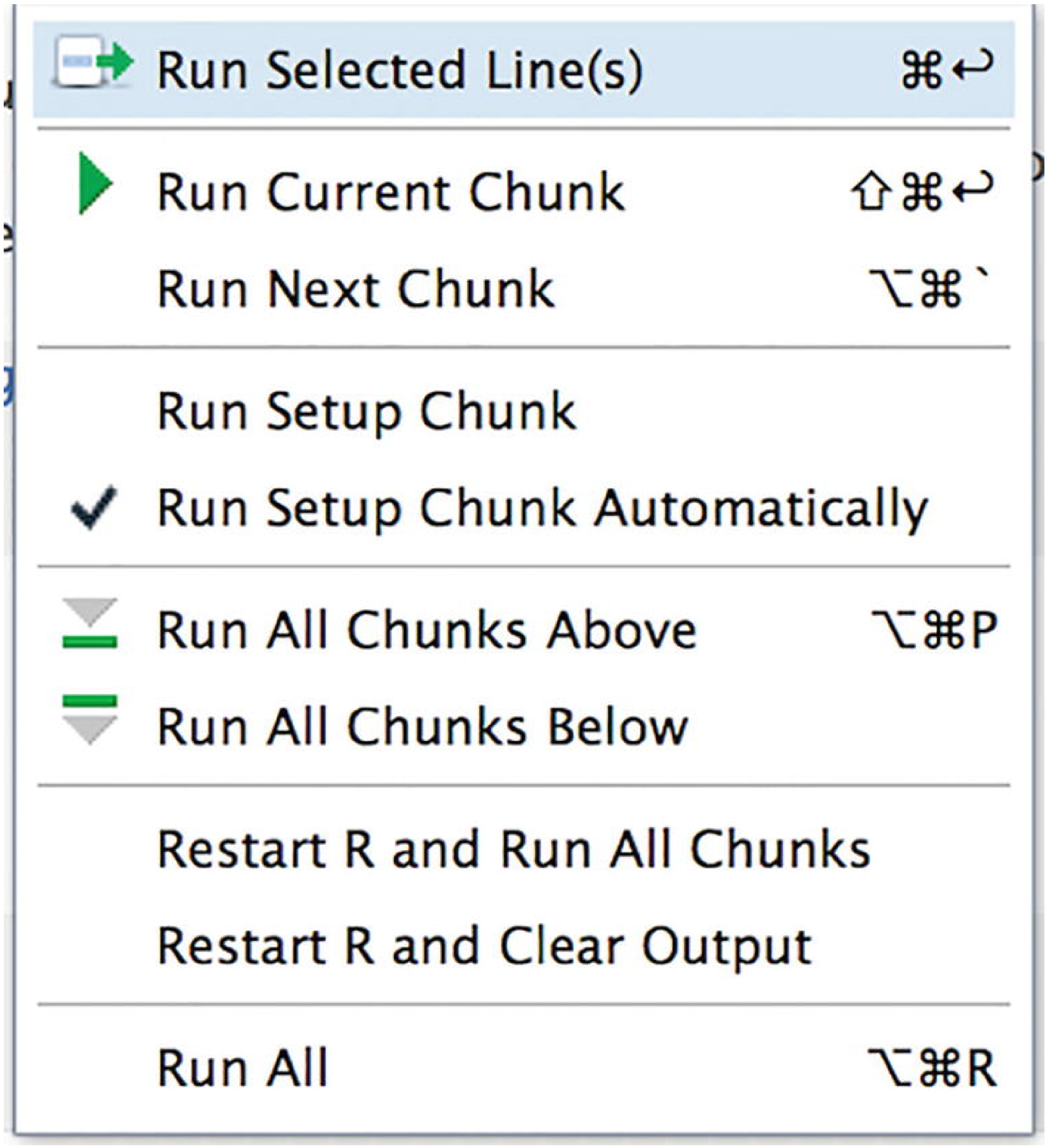


Figure 2-8 Options for evaluating chunks

Even without the option for generating final documents from a Markdown document, I would still be using them just for this feature.

Caching Results

Sometimes, part of an analysis is very time-consuming. Here, I mean in CPU time, not thinking time—it is also true for thinking time, but you don't need to think the same things over again and again. If you are not careful, however, you will need to run the same analysis on the computer again and again.

If you have such very time-consuming steps in your analysis, then compiling documents will be very slow. Each time you compile the document, all the analysis is done from scratch. This is the functionality you want since this makes sure that the analysis does not have results left over from code that isn't part of the document, but it limits the usability of the workflows if they take hours to compile.

To alleviate this, you can cache the results of a chunk. To cache the result of a chunk, you should add the option `cache=TRUE` to it. This means adding that in the header of the chunk similar to how output options are added. You will need to give the chunk a name to use this. Chunks without names are actually given a default name, but this name changes according to how many nameless chunks you have earlier in the document, and you can't have that if you use the name to remember results. So you need to name it. A named chunk that is set to be cached will not only be when you compile a document if it has changed since the last time it was evaluated. If it hasn't been changed, the results of evaluating it will just be reused.

R cannot cache everything, so if you load libraries in a cached chunk, they won't be loaded unless the chunk is evaluating, so there are some limits to what you can do, but generally it is a very useful feature.

Since other chunks can depend on a cached chunk, there can also be problems if a cached chunk depends on another chunk, cached or not. The chunk will only be reevaluated if you have changed the code inside it, so if it depends on something you have changed, it will remember results based on outdated data. You have to be careful about that.

You can set up dependencies between chunks, though, to fix this problem. If a chunk is dependent on the results of another chunk, you can specify this using the chunk option `dependson=other`. Then, if the chunk `other` (and you need to name such chunks) is modified, the cache is considered invalid, and the depending chunk will be evaluated again.

Displaying Data

Since you are writing a report on data analysis, you naturally want to include some results. That means displaying data in some form or other.

You can simply include the results of evaluating R expressions in a code chunk, but often you want to display the data using tables or graphics, especially if the report is something you want to show to people not familiar with R. Luckily, both tables and graphics are easy to display.

To make a table, you can use the function `kable()` from the `knitr` package. Try adding a chunk like this to the boilerplate document you have:

```
library(knitr)
kable(head(cars))
```

The `library(knitr)` imports functions from the `knitr` package so you get access to the `kable()` function. You don't need to include it in every chunk you use `kable()` in, just in any chunk before you use the function—the `setup` chunk is a good place—but adding it in the chunk you write now will work.

The function `kable()` will create a table from a data frame in the Markdown format, so it will be formatted in the later step of the document compilation. Don't worry too much about the details about the code here; the `head()` function just picks out the first lines of the `cars` data so the table doesn't get too long.

Using `kable()` should generate a table in your output document. Depending on your setup, you might have to give the chunk the output option `result="asis"` to make it work, but it usually should give you a table even without this.

We will cover how to summarize data in later chapters. Usually, you don't want to make tables of full data sets, but for now, you can try just getting the first few lines of the `cars` data.

Adding graphics to the output is just as simple. You simply make a plot in a code chunk, and the result will be included in the document you generate. The boilerplate R Markdown document already gives you an example of this. We will cover plotting in much more detail later.

Exercises

Create an R Markdown Document

Go to the File… menu and create an R Markdown document. Read through the boilerplate text to see how it is structured. Evaluate the chunks. Compile the document.

Different Output

Create from the same R Markdown document an HTML document, a document, and a Word document.

Caching

Add a cached code chunk to your document. Make the code there sample random numbers, for example, using `rnorm()`. When you recompile the document, you should see that the random numbers do not change.

Make another cached chunk that uses the results of the first cached chunk. Say, compute the mean of the random numbers. Set up dependencies and see that if you modify the first chunk the second chunk gets evaluated.

Footnotes

1 https://jupyter.org

2 https://en.wikipedia.org/wiki/Javadoc

3 http://roxygen.org

4 www.wolfram.com/mathematica

5 http://jupyter.org

6 https://bookdown.org/yihui/bookdown

7 In any case, having cross-references to sections but not figures is still better than Word where the feature is there but buggy to the point of uselessness, in my experience…

8 http://rmarkdown.rstudio.com/authoring_bibliographies_and_citations.html

9 http://yihui.name/knitr/
