Chapter 7: Reproducible Data Analysis

Mark Andrews

Contents

Introduction **1**

Using RMarkdown for reproducible reports **3**
- Installation . . . 4
- A minimal RMarkdown example . . . 5
- An extended RMarkdown example . . . 6
- An additional RMarkdown example . . . 10
- Brief guide to Markdown . . . 16
- Brief guide to mathematical typesetting with LaTeX . . . 20

Git **25**
- Installation . . . 26
- Configuration . . . 26
- Creating and initiating a Git repository . . . 27
- Adding and editing files . . . 31
- Using remote repositories . . . 34

References **36**

Introduction

The end product of any data analysis is usually a set of tables, figures, and the seemingly countless statistics and other quantities that specify the results of the statistical modelling or testing and that was performed. These results are then usually communicated in reports, including and especially peer-reviewed scientific articles, or through talks and presentations, or through mainstream or social media, and so on. These results are the end product of an often long and arduous process, something we will call the *data analysis pipeline*, that began initially with just the raw data. For example, the original raw data might have been in the form of a set of `.xlsx` files that were downloaded from a website. These might have been wrangled and transformed repeatedly by a long series of operations, such as those described in Chapter 3, to produce various "tidy" data-sets, which themselves might then have been repeatedly and iteratively visualized and statistically analysed and modelled. It is not an exaggeration to say that this whole process might have involved hundreds or even thousands of hours of work, taking place intermittently over the course of months or even years, and involving many different people at many different stages. We can view the data analysis pipeline as akin to a factory: raw materials, in the form of the raw data, go in; these are worked on and turned into something new and valuable by the combined efforts of human labour and machines (computers); and finally the end products are produced that are to be consumed by others.

The aim of *reproducible data analysis*, at its most general, is to make the data analysis factory or pipeline as open and transparent as possible, and to allow all others, including our future selves, to be able to exactly reproduce any of the results that were produced by it. In other words, the aim is to make it possible for anyone to determine exactly where and how any given table, figure, or statistical quantity was obtained, and to be able to reproduce any of these results exactly. If the pipeline is suitably reproducible, anyone ought to

be able to take the original raw data and reproduce all the final results, or alternatively ought to be able to take the raw data and analyse them in new and different ways, thus producing new and interestingly different results.

Doing reproducible data analysis is often motivated by a general commitment to doing *open science*. Since the origin of modern science in the 17th century, part of its defining ethos (see Merton 1973) has been the unrestricted sharing of the fruits of research, and also a full disclosure of all details of any research so that they may be scrutinized by others. Thus, openness and transparency are core ethical principles in all science. Recently, it has become apparent that these ethical principles, although often endorsed in principle, are not usually followed in practice, and that in fact there is a widespread culture of not sharing data (see, for example, Tenopir et al. 2011; Houtkoop et al. 2018; Fecher, Friesike, and Hebing 2015), not sharing data analysis and other computer code (see, for example, Stodden, Guo, and Ma 2013; Shamir et al. 2013), and that there is a widespread general lack of research transparency in science (see, for example, Iqbal et al. 2016). This has lead to repeated calls for major cultural changes related to the sharing of data and code and general research transparency under the banner of doing *open science* (for example, Nosek et al. 2015; Munafò et al. 2017; Gorgolewski and Poldrack 2016; Ioannidis 2015).

Reproducible data analysis can also be motivated simply as a means of doing more high quality and robust data analysis, even when this analysis is not being done as part of scientific research per se, such as with confidential analyses that is done in business and industry. In these contexts, the raw data and analysis pipeline may always remain confidential and never be shared publicly. Nonetheless, doing this analysis using reproducible research practices allows for essential quality control. It allows the analysts themselves, those they do the analysis for, or future analysts who inherit the project or are brought on board it later, to scrutinize and double-check every detail of the analysis and reproduce every result. These are essential measures to identify errors, increase rigour, and verify the final results and conclusions. Even, or especially, outside of academic or scientific research, there can be enormous practical and financial incentives to minimizing errors and increasing analytical rigour in data analysis. As an example, it has been argued that a lack of reproducible data analysis techniques was partly to blame for a $9bn loss at the investment bank JPMorgan in 2012 (see Hern 2013).

This chapter is about doing *open*, *transparent*, and *reproducible* research using R and related software tools. In particular, we will focus on *RMarkdown* and *knitr* for making reproducible reports, and *Git* for version control of all our files. There are other important software tools for reproducible research that we could cover, but do not do so in order to keep our coverage relatively brief and introductory. These tools include *Docker* for creating lightweight virtual operating systems, R packages for packaging and distributing code and data, build automation tools like *GNU Make* or the R package `drake` (Landau 2018), continuous integration tools like *Jenkins* (https://www.jenkins.io/) or *Travis CI* (https://travis-ci.com/).

Before we proceed, however, let us briefly discuss some terminology. The terms *open*, *transparent*, and *reproducible* are a collection of adjectives that describe a set of data analysis practices. These terms are obviously not identical, but they are related. It is, however, not trivial to state the extent to which any one depends upon or requires any other. While it is not necessary to be pedantic about the definitions and scope of these terms, we will briefly outline our understanding of each term.

Open Open data analysis, like open source software or open science generally, is data analysis where all the data, code, and any other required materials are fully disclosed to and shared with others, usually by being made publicly available. Being publicly available, and in an unrestricted manner, is therefore usually taken to be a defining feature of open data analysis.

Transparent Transparent data analysis is analysis where, as mentioned above, it is possible to determine exactly where and how any given table, figure, or statistical result was obtained. Making data and code and all other material open is usually a sufficient condition for the analysis to be transparent, but it is possible to conceive of situations where data and code are open and available but are poorly written and organized, or are obfuscated, or require undocumented manual intervention, and so lack transparency. Likewise, as also mentioned above, it is not necessary for data to be open, at least in the sense of publicly available, for it to be transparent.

Reproducible Reproducible data analysis is any data analysis where an independent analyst can exactly reproduce all the results. For an analysis to be reproducible it is necessary that all the data and code is available and in full working condition. Strongly implied, however, is that running the code is essentially a turnkey operation. Software tools, such as *build automation* tools like *GNU make*, are often used, especially with larger projects. For reports and articles, and their myriad tables and figures and in-text quantities, *literate programming* tools, particularly `rmarkdown` with R, are used. Version control software, such as `git`, is often used to organize and keep a log of the development of all the analysis code and scripts. To allow the code to be used across different operating systems and with the correct dependencies, virtual or containerized operating systems using tools such as `docker` are sometimes used. Other tools, such as the `checkpoint::checkpoint()` function in R can be used to ensure the correct versions of R package dependencies are used.

Using RMarkdown for reproducible reports

As mentioned, the results of data analyses are communicated through reports such as peer reviewed scientific articles and other manuscripts, slide or poster presentations, webpages, and so on. For the results described in these reports to be truly reproducible, every figure, table, or any other value or quantity ought to be reproducible with minimal effort by anyone, including and especially those not directly involved in the analysis. For this, it is necessary that the data and code be made available to others. But this alone is not sufficient. Even if the necessary data and code were complete, it might still be challenging or nearly impossible for someone who was not directly involved in the analysis to identify exactly which code produced any given figure, table, or other reported quantity. This is especially the case if the results in the report were generated, as is exceedingly common, by transcribing or copying and pasting results from the output of statistical software, or inserting figures that were exported, into a document such as MS Word or LaTeX. For any given figure or table or quantity, hunting down and verifying which piece of code produced it could be very challenging and time consuming. Moreover, there is a very high probability that the reported results will have some errors somewhere simply because they were all the product of transcription or other manual interventions. Indeed, there is no way of even categorically verifying that all the correct code and data necessary to produce the results has been made available without painstakingly finding and checking each individual piece code for each individual result.

For a report to be truly reproducible, therefore, the data, code, and resulting document need to be inextricably coupled in a manner that goes beyond how documents are traditionally written. This coupling of code, data, and text is based on the concept of *literate programming* (see Knuth 1984). In a *literate program*, the text of the documentation of a computer program and the code of that program are linked in one single file or set of files. These files can be processed by different programs either to generate the documentation manuals or the computer programs that will be compiled and executed like normal programs. While this principle has become a common means of generating documentation for computer code (see, for example, *Roxygen2* for documenting R code, *Doxygen* for documenting C++, *Sphinx* for documenting Python, etc), the same principle can be used to creating reproducible data analysis reports, which are sometimes known as *dynamic documents* (see Xie 2017) The most popular, and arguably the best, way of doing this using R is to use *RMarkdown*, or more precisely to use RMarkdown and a combination of tools including *knitr*, *pandoc*, LaTeX, and others.

Before we proceed, let us briefly introduce or define a number of key concepts and tools.

RMarkdown RMarkdown itself is simply a text file format, and so an RMarkdown file is essentially a script, not unlike an R script. In fact, just like normal R scripts, we usually open, edit, and "run" RMarkdown files inside of RStudio. Unlike R scripts, which consist of just R code, possibly including comments, but nothing else, RMarkdown scripts primarily consist of a mixture of two type of code: *Markdown* code and normal R code. As we'll see, the R code in RMarkdown documents occurs in either R *chunks*, which are simply blocks of code, or in *inline* R code, which are small amounts of R code inside Markdown code.

Markdown Markdown is a minimal or lightweight markup language. It was initially created by John Gruber

(Gruber 2004) primarily as means of allowing web users to create formatted posts, including links, images, lists, etc., on online discussion forums. Markdown consists of normal text, just as you would write in an email or any other document, as well some minimal syntax that instructs how this text should be formatted when it is rendered into some output document, such as a html page, pdf or Word document.

Knitr Knitr (see Xie 2017) is a general tool for dynamic documentation generation using R. In brief and put very simply, knitr extracts R code from RMarkdown documents, runs this code, and then, by default, inserts copies of this code and the code's output into a Markdown document. Knitr then runs *pandoc* to create a document, such as a pdf, MS Word document, etc, from this Markdown file.

Pandoc Pandoc (MacFarlane 2006) is, in general, a document converter that can convert a large number of input document types to an equally large number of output document types. For present purposes, it is the means by which Markdown documents generated by knitr are converted into their final output formats such as pdf, MS Word, html, etc.

LaTeX LaTeX [@lamport1994latex] is a document preparation system that is specialized for creating high quality technical and scientific documents, especially those containing mathematical formulas and technical diagrams. It is widely used for creating academic and research manuscripts in mathematically oriented fields such as statistics, computer science, physics, etc. LaTeX documents are created by first writing a `.tex` source code file, which is mixture of LaTeX and TeX code, markup syntax, and plain text. This `.tex` file is then rendered to, usually, a pdf using a LaTeX rendering engine, of which there are many but the most widely is the default `pdflatex` engine.

Installation

If using RStudio, the necessary R packages including `rmarkdown` and `knitr` are automatically installed. Likewise, the external pandoc program is also automatically installed when `rmarkdown` is installed. While this set of tools will allow you to create html and MS Word documents, LaTeX must be installed to create pdf outputs. Installation and configuration of LaTeX, because it is a large external program, may not always be straightforward. It may seem, therefore, that this LaTeX installation step is not worthwhile, especially given that many people unquestionably and happily use MS Word to write their manuscripts. However, we *highly* recommend installing LaTeX and using the LaTeX-ed pdf document output over MS Word for writing manuscripts. At the very least, the typesetting quality of the resulting LaTeX-ed document will be considerably higher than that of the MS Word. Moreover, by using LaTeX with RMarkdown, we can avail of all the power of LaTeX to create the final document. This includes the use of all LaTeX packages for creating and styling mathematical and technical content, including technical diagrams, fine control of figures and tables, their look and feel, and their placement, internal cross referencing to refer to individual pages, sections, figures, etc., automatic index generation, and many more. Even if these features may seem unnecessary at the beginning, in our opinion, it is nonetheless still worthwhile to use LaTeX with RMarkdown from the beginning, so that all these features can be used as one's experience with RMarkdown grows.

To install LaTeX for the purposes of using with RMarkdown, we can install the R package `tinytex`.

```
install.packages("tinytex")
tinytex::install_tinytex()
```

This installation will take some time. After it completes, you should restart RStudio. After you restart, type the following command (note the three `:`'s after `tinytex`):

```
tinytex:::is_tinytex()
```

After this installation completes, we can test that `rmarkdown` (which will have be installed by `tidyvese`) will render pdf documents using LaTeX with the following code.

```
writeLines("Hello $x^2$", 'test.Rmd')
rmarkdown::render("test.Rmd", output_format = "pdf_document")
```

The `writeLines` creates a tiny `.Rmd` file named `test.Rmd`. The `rmarkdown::render` command then attempts to render this as pdf document, which will require LaTeX etc be properly working. If this works as expected, we will have a pdf document named `test.pdf` in our working directory.

A minimal RMarkdown example

As just mentioned, an RMarkdown file is essentially a script containing two types of code: Markdown and normal R code. The following is the contents of small RMarkdown file, named `example.Rmd`.

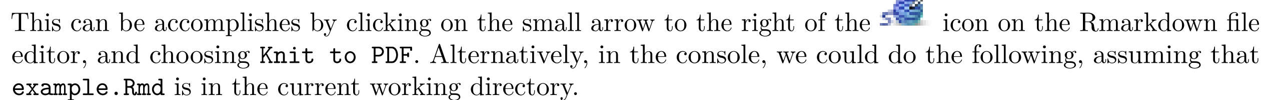
~~~~
This is some text. Some of it is *italicized*, and some is in **bold**.

```{r}
x <- c(5, 12, 103)
y <- x ^ 2
```

The mean of `x` is `r mean(x)`, and the min value of `y` is `r min(y)`.
~~~~

This example contains one R *chunk*. This is the portion of R code between the opening ` ```{r} ` and the closing ` ``` `. It also has two pieces of inline R code. These are the code segments between the opening `` `r `` and closing `` ` ``. The remainder of the code in this example is plain Markdown code.

As we will see, we could open this file in the RStudio editor and then *knit* it to, for example, a pdf document.



This can be accomplishes by clicking on the small arrow to the right of the icon on the Rmarkdown file editor, and choosing `Knit to PDF`. Alternatively, in the console, we could do the following, assuming that `example.Rmd` is in the current working directory.

A pdf document `example.pdf` will be produced and its contents will be as follows.

This is some text. Some of it is *italicized*, and some is in **bold**.

```
x <- c(5, 12, 103)
y <- x ^ 2
```

The mean of `x` is 40, and the min value of `y` is 25.

The key features to notice here are that the Markdown code within the RMarkdown file is formatted according to its instructions. For example, the appropriate words are italicized and emboldened. Also, the R code in the chunk is appropriately formatted as code, using a monospaced font, and is syntax highlighted. Finally, and most interestingly in this example, the *values* returned by the R commands in the two pieces of inline code are inserted at the locations of, and thus replace, the original two pieces of inline code. In other words, in the final document, the values of `40` and `25` replace `` `r mean(x)` `` and `` `r min(y)` ``, respectively.

Although this is a very simple example, a lot has happened to produce the final pdf output. The procedure is roughly as follows:

- First, `knitr` extracts out the R code from `example.Rmd` and runs them in a separate R session. The code is run in the order in which it appears in the document. So, first, the R chunk is run, which creates two vectors, `x` and `y`. Then, the R commands `mean(x)` and then `min(y)` are executed.
- Next, all the Markdown code in `example.Rmd` is extracted out and inserted into a new temporary file `example.md`. Likewise, by default, copies of the R code in the chunk, though not the inline code, are inserted into this `example.md` Markdown file. Any R code that is inserted into `example.md` is marked-up as R code so that it will be properly rendered, including with syntax highlighting, in the final output document.

- Next, any *output* from the R code, whether the R chunk or the R inline code is inserted into the `example.md` file. In the example above, there is no output from the R chunk. However, obviously the commands `mean(x)` and `min(y)` both return numbers, and so these two numbers are inserted into the `example.md` Markdown file at the exact locations of where the two pieces of inline R code occurred.
- Now, *knitr* calls *pandoc*. Pandoc firsts convert the `example.md` Markdown into a LaTeX source code file named `example.tex`. This file is essentially just another script whose contents would be easily understandable to anyone who knows LaTeX.
- Next, pandoc calls a LaTeX rendering engine, such as `pdflatex`, and converts the LaTeX source code file `example.tex` into the pdf document `example.pdf`.
- Finally, the intermediate files, including `example.md` and `example.tex` are, by default, removed so that the only remaining files are the original RMarkdown source file `example.Rmd` and the final output document `example.pdf`.

An extended RMarkdown example

````
---
title: "Data Analysis: A Report"
author: "Mark Andrews"
date: "October 25, 2019"
output: pdf_document
---

```{r setup, echo=FALSE}
knitr::opts_chunk$set(message = F,
                      warning = F,
                      out.width = "45%",
                      fig.align='center')
```

# Introduction

First, we will load the `tidyverse` packages, and read
in the the from a `.csv` file.
```{r load_packages_data}
library(tidyverse)
data_df <- read_csv('example.csv')
```

# Analysis

Here, we do a Pearson's correlation analysis.
```{r analysis}
(corr_model <- cor.test(~ x + y, data = data_df))
```

The correlation coefficient is
`r round(corr_model$estimate, 3)`.

# Visualization

The scatterplot between $x$ and $y$ is shown
in Figure \ref{fig:vis}.

```{r vis, echo=F, fig.cap='A scatterplot.'}
ggplot(data_df, aes(x, y)) +
  geom_point() +
  theme_classic()
```
````

Data Analysis: A Report

Mark Andrews

October 25, 2019

Introduction

First, we will load the `tidyverse` packages, and read in the the from a `.csv` file.

```
library(tidyverse)
data_df <- read_csv('example.csv')
```

Analysis

Here, we do a Pearson's correlation analysis.

```
(corr_model <- cor.test(~ x + y, data = data_df))
```

```
##
##  Pearson's product-moment correlation
##
## data:  x and y
## t = 2.1583, df = 48, p-value = 0.03593
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.02080288 0.53175304
## sample estimates:
##       cor
## 0.2974284
```

The correlation coefficient is 0.297.

Visualization

The scatterplot between x and y is shown in Figure 1.

1

Figure 1: An example of a RMarkdown file on the left, and its corresponding pdf output on the right. The pdf output is generated by *knitting* the RMarkdown file. In RStudio, this can be accomplished by `rmarkdown::render("example-2.Rmd")`, where `example-2.Rmd` is the RMarkdown file.

In Figure 1, we shown the code of an RMarkdown file on the left and its corresponding pdf output on the right. This example shows many of the features of a typical RMarkdown document. We will discuss these features by looking through each section of the RMarkdown code.

The YAML header

The first few lines of the document, specifically those lines that are delimited by the lines with the three dashes (i.e., `---`) constitute its *YAML* header.

```
---
title: "Data Analysis: A Report"
author: "Mark Andrews"
date: "October 25, 2019"
output: pdf_document
---
```

YAML (a recursive acronym for *Y*AML *A*in't a *M*arkup *L*anguage) is itself a minimal markup language that is now often used for software configuration files. As is perhaps clear from this example, in this header we specify the title, author, date, and output format of the document. This information is used when rendering the output document. At its simplest, YAML consists of *key-value* mappings where the term on the left of the `:` is the key, and the term or statement on the right is the value. Thus, `author: "Mark Andrews"` indicates that the `author` is `"Mark Andrews"`, and when the final document is being generated, the `"Mark Andrews"` will be inserted as the value of the author in output document's template. Note that although we use use double quotation marks as the values of the `title`, `author`, and `date` keys, neither single nor double quotation marks are always necessary. In this example, we would only require the quotations for the value of `title`. This is to ensure that the colon within `Data Analysis: A Report` is not mistakenly parsed by the YAML parser as indicating a key-value mapping.

We specify the document output type in the header by `output: pdf_document`. As mentioned above, when writing manuscripts, we recommend always using the pdf based output document option as this will be generated by LaTeX. However, had we preferred to create an MS Word output document, we would state `output: word_document`. Likewise, had we preferred a HTML output document, which can obviously be viewed in any browser or used to create a webpage, we would do `output: html_document`.

The setup R chunk

It is common, though not necessary to have a setup R chunk like the following at the beginning of our RMarkdown file.

````
```{r setup, echo=FALSE}
knitr::opts_chunk$set(message = F,
                      warning = F,
                      out.width = "45%",
                      fig.align='center')
```
````

In addition to including a common configuration statement, this example also has some important general RMarkdown features. First, in the minimal RMarkdown example in the previous statement, the R chunk was defined by starting with the opening `` ```{r} `` and ending with the closing `` ``` ``. Here, the chunk statement begins with

````
```{r setup, echo=FALSE}
````

The term `setup` after the initial `r` is simply a label for this chunk. It is not necessary to have a label for chunks but it can be used for file navigation purposes in the RStudio editor, and is used for cross referencing purposes, especially for so-called *floating* figures and tables, as we will see. As such, it is probably a good habit to always use a (unique) label for each R chunk.

After the chunk label, we have the chunk configuration option `echo=FALSE`. This indicates the the R code in this chunk should not be displayed in the output document. Next, we have the following R statement.

```
knitr::opts_chunk$set(message = F,
```
```

```
                  warning = F,
                  out.width = "45%",
                  fig.align='center')
```

Here, we are set the values of `opts_chunk` list of global options used by `knitr`. For example, in this case, we start by indicating that, by default, all subsequent chunks in the RMarkdown document should have the options: `message = FALSE` and `warning = FALSE` (because these are R statements, we can use `F` for `FALSE`). Setting `message = FALSE` entails that the messages, which can be verbose, produced by R commands are not shown. As an example of this, consider the usual output when we load a package, in this case using `lme4`.

```
library(lme4)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

Therefore, globally setting `message = FALSE` can considerably reduce clutter in the output document. Likewise, it is often preferable to globally suppress R warnings to avoid clutter in the final document. Although not used in this example, another commonly set global option at this point is `echo=FALSE`. Often, especially in manuscripts for peer-reviewed articles, we do not wish to display all, or even any, of the R code used in the analysis. Rather, we just want to display the results of the analysis in tables, figures, etc, and so globally setting `echo=FALSE` avoids having to set `echo=FALSE` on each subsequent chunk.

The next two settings, `out.width = "45%"` and `fig.align='center'`, pertain to how figures should be displayed by default. The `out.width = "45%"` indicates that the figure should occupy 45% of the width of the page, and `fig.align='center'` indicates that it should be centered on the page.

Markdown sections

The use of the `#` followed by some text at the start of a line in the Markdown language indicates a section header. Thus, in our case, the following line indicates we should have a section entitled *Introduction* in our document.

```
# Introduction
```

While the single `#` is for a section, subsections are indicated by `##`, and subsubsections are indicated by `###`. Here is an example with three levels of sections.

```
# This is a section

## This is a subsection

# This is another section

## This is a subsection

### This is a subsubsection
```

In principle, further nested subsections, e.g. subsubsubsections and so on, are possible. However, this does depend on both the output document type and also the templates for this documents that are used by `knitr` or pandoc.

R chunk outputs

Just as we see the output of R commands when those commands are run in the console, the output of R commands will appear in the output document unless we specify request otherwise. Consider the following lines from our example RMarkdown file.

````
```{r analysis}
(corr_model <- cor.test(~ x + y, data = data_df))
```
````

In this case, the assignment statement is surrounded by `(` and `)` and in general in R, this causes the output of the expression in this statement to be shown. In other words, the output that will be shown will be identical to what would be shown had be just typed.

```
cor.test(~ x + y, data = data_df)
```

By default, this output will have each line beginning with `##` (because they are part of the R chunk's output are not interpreted as subsection headers). Should we wish use alternative symbols at the beginning of the output, we can indicate this by setting the value of the `comment` chunk option. For example, if we wanted all R output to start with '>', we set `comment = '>'`, as in the following code.

````
```{r, echo=T, comment='>'}
rnorm(5) %>% round(2)
```
````

This code would then be rendered as follows.

```
rnorm(5) %>% round(2)
```

```
> [1]  0.54  1.12 -1.40 -3.21  2.02
```

LaTeX mathematical typesetting

On the following line

```
The scatterplot between $x$ and $y$ is shown
```

we have the terms `$x$` and `$y$`. The dollar symbols indicate that `x` and `y` should be parsed using LaTeX and typeset according. In this example, this will simply make the `x` and `y` be shown in a different and italicized font compared to normal letters. However, in general, there is a vast number of mathematical notations, formulae, and symbols that can be used here. For example, consider the following Markdown code.

```
If $\Phi = \phi_1, \phi_2 \ldots \phi_k \ldots \phi_k$, where each $0 \leq \phi_k \leq 1$,
and $\sum_{k=1}^K \phi_k = 1$, then $\Phi$ is a probability mass function.
```

This code would then be rendered as follows.

If $\Phi = \phi_1, \phi_2 \ldots \phi_k \ldots \phi_k$, where each $0 \leq \phi_k \leq 1$, and $\sum_{k=1}^K \phi_k = 1$, then Φ is a probability mass function.

In these examples, all the LaTeX code occurs in its *inline mode*, and this obtained by using `$` delimiters. In addition to inline mode, there is LaTeX *display* mode. This is obtained by using `$$` delimiters. LaTeX display mode is used to display mathematical formulae and other notation on newlines. Consider the following example.

```
The probability of the observed data is as follows:
$$
\mathrm{P}(x_i \ldots x_n \vert \mu, \sigma^2)
= \prod_{i=1}^n \mathrm{P}(x_i \vert \mu, \sigma^2),
$$
```

```
where $\mu$ and $\sigma$ are parameters.
```

This would then be rendered as follows.

The probability of the observed data is as follows:

$$\mathrm{P}(x_i \ldots x_n|\mu, \sigma^2) = \prod_{i=1}^{n} \mathrm{P}(x_i|\mu, \sigma^2),$$

where μ and σ are parameters.

LaTeX provides a very large number of mathematical symbols, notations, and formulae. It is beyond the scope of this chapter to even provide an overview of all of these features. However, we will provide an overview of some of the more widely used examples in a later section of this chapter.

Figures

If the R code in the a chunk generates a figure, for example by using `ggplot`, then that figure is inserted into the document immediately after the location of the chunk. These figures can optionally have figure captions by setting the value of `fig.cap` in the chunk header. When the `fig.cap` is set, at least when using the `pdf_document` output, the figures then *floats*. A floating figure is one whose placement location in the final document is determined by an algorithm that attempts to minimize a large number of constraints, such as the amount of whitespace on the page, the number of figures on one page, and so on.

In our example file, we have the following lines.

````
```{r vis, echo=F, fig.cap='A scatterplot.'}
ggplot(data_df, aes(x, y)) +
  geom_point() +
  theme_classic()
```
````

This chunk will create a `ggplot` figure that will float. Because of the global settings of `out.width = 40%` and `fig.align = 'center'`, the figure will be relatively small and centered on the page. Because there is sufficient space, the figure will be placed at the bottom of the page. However, were we to increase the width of the figure to even `out.width = 50%`, the default settings of the float placement algorithm would place the figure alone on the following page. Often, the choices made by LaTeX's float placement algorithm are initially unappealing. The recommendation, however, is to not attempt any fine control until the document is complete. As each new text block, or figure, or table is added to the document, the floats are relocated. When the document is complete, it is possible to influence float placement by changing parameter settings on the float placement algorithm.

It should be noted that if figures do not float, they are placed exactly where the chunk's output would otherwise occur. In some cases, especially if the figures are relatively small, this can be satisfactory.

An additional RMarkdown example

In Figure 2, we provide the code of another RMarkdown file. This example provides some additional important RMarkdown features that were not covered in the previous example. The pdf output document corresponding to this document is shown in Figure 3.

Additional YAML header options

In this example file, we have set some additional settings in the YAML header. For example, we have the following lines.

```
---
title: "Data Analysis: Report II"
author: "Mark Andrews"
date: "October 25, 2019"
output:
  pdf_document:
    keep_tex: yes
header-includes:
- \usepackage{booktabs}
bibliography: refs.bib
biblio-style: apalike
---

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

```{r load_packages, echo = FALSE}
library(tidyverse)
library(knitr)
library(kableExtra)

weight_df <- read_csv('weight.csv')
weight_df_grouped <- group_by(weight_df, gender)

sample_size <- weight_df_grouped %>%
  summarise(n=n()) %>%
  deframe()

weight_df_summary <- weight_df_grouped %>%
  summarise_at(vars(weight, height),
               list(avg=mean, stdev=sd))
```

Descriptive statistics

In this data set, we have measured the weight (in kg)
and height (in cm) of `r sum(sample_size)` participants
(`r sample_size['Male']` males, `r sample_size['Female']`
females). In the following table, we show the mean and
separately for males and females.

```{r descriptives_table, echo=F}
weight_df_summary %>%
  kable(format = "latex",
        booktabs = TRUE,
        digits = 2,
        align = 'c') %>%
    kable_styling(position = "center")
```

Statistical model

We will model the relationship between weight and height
as a varying intercepts normal linear model as follows.
For each $i \in 1 \ldots n$,
$$
\begin{aligned}
y_i &\sim N(\mu_i, \sigma^2),\\
\mu_i &= \beta_0 + \beta_1 x_i + \beta_2 z_i,
\end{aligned}
$$
where y_i, x_i, z_i are the weight, height,
and gender of participant i.

In R, this analysis can be easily performed as follows.
```{r stat_model}
model <- lm(weight ~ height + gender, data = weight_df)
```
```{r model_results, echo=F}
R_sq <- summary(model)$r.sq
f_stat <- summary(model)$fstatistic
p_value <- pf(f_stat[1], f_stat[2], f_stat[3], lower.tail = F)
```

The R^2 for this model is `r round(R_sq, 2)`,
$F(`r round(f_stat[2])`, `r round(f_stat[3])`) =
`r round(f_stat[1],2)`$,
$p `r format.pval(p_value, eps = 0.01)`$.

More information about varying intecept models can be
found in @GelmanHill:2007.

References
```

Figure 2: An additional RMarkdown file example. The rendered pdf output document is shown in Figure 3.

```
output:
  pdf_document:
    keep_tex: yes
header-includes:
- \usepackage{booktabs}
```

The latter two lines relate to the bibliography, and we will return to these below. The first two lines indicates that the LaTeX command `\usepackage{booktabs}` should be included in the header of the resulting `.tex` file prior to be rendered to pdf by the LaTeX engine. Recall that as mentioned above, the rendering process to produce the final `.pdf` document is that `knitr` takes the `.Rmd` file and creates a `.md` file, with R code and R code output inserted into it. If the final output is to be pdf, pandoc converts the `.md` file to a `.tex` file, and that is then rendered to `.pdf` by a LaTeX engine such as `pdflatex`. When pandoc creates the `.tex` file from the `.md`, it uses `.tex` file templates that load some of the more widely used LaTeX packages. However, if any other LaTeX packages are required to create the final pdf document, they may be listed as we have done here in the YAML header using the `header-includes` option. It should be noted that there are many thousands of additional LaTeX packages that can be installed.

Another option to include addition LaTeX functionality in your RMarkdown is to include a block of LaTeX code into the header. For example, we might have a file named `include.tex` with the following LaTeX command.

```
\newcommand{\Prob}[1]{\mathrm{P}( #1 )}
```

If we had the following YAML header in our RMarkdown document, the commands in `include.tex` would

Data Analysis: Report II

Mark Andrews

October 25, 2019

Descriptive statistics

In this data set, we have measured the weight (in kg) and height (in cm) of 6068 participants (4082 males, 1986 females). In the following table, we show the mean and the standard deviations of the weights and heights, separately for males and females.

gender	weight_avg	height_avg	weight_stdev	height_stdev
Female	67.76	162.85	10.98	6.42
Male	85.52	175.62	14.22	6.86

Statistical model

We will model the relationship between weight and height as a varying intercepts normal linear model as follows. For each $i \in 1 \ldots n$,

$$y_i \sim N(\mu_i, \sigma^2),$$
$$\mu_i = \beta_0 + \beta_1 x_i + \beta_2 z_i,$$

where y_i, x_i, z_i are the weight, height, and gender of participant i.

In R, this analysis can be easily performed as follows.

```
model <- lm(weight ~ height + gender, data = weight_df)
```

The R^2 for this model is 0.45, $F(2, 6065) = 2495.83$, $p < 0.01$.

More information about varying intecept models can be found in Gelman and Hill (2007).

References

Gelman, Andrew, and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models.* New York: Cambridge University Press.

Figure 3: The pdf document output corresponding to the RMarkdown code shown in Figure 2.

be available to use in the body of the RMarkdown file.

```
output:
  pdf_document:
    includes:
      in_header: include.tex
```

For example, in this case, we could now write `\Prob{X = x} = 0.25` in our RMarkdown file, and this would eventually be rendered as $\mathrm{P}(X = x) = 0.25$. The file `include.tex` can have an extensive number of custom commands, including numerous `\usepackage{}` statements.

Formatted tables

When writing reports, we often wish to present results of statistical analyses in tables. Naturally, we would ideally like these tables to be formatted to a high standard. As we've seen, we can always simply display the standard output of R commands that produce tables and data frames, but the resulting unformatted monospaced font text is not of an acceptable standard for most manuscripts and reports, especially those that are intended for eventual publication. As an example, consider the table produced by the following R commands that use the built in `swiss` data set.

```
(swiss_df <- swiss %>%
  select(Fertility:Examination) %>%
  slice(1:5))
```

```
##              Fertility Agriculture Examination
## Courtelary        80.2        17.0          15
## Delemont          83.1        45.1           6
## Franches-Mnt      92.5        39.7           5
## Moutier           85.8        36.5          12
## Neuveville        76.9        43.5          17
```

Clearly, this output is not sufficient for all but informal reports.

Fortunately, we have a number of options for formatting tables. For example, to use `kable` to create a table specifically for use in the `pdf_document` output, we can do the following.

```
library(knitr)
swiss_df %>%
  kable(format = 'latex',
        booktabs = TRUE,
        align = 'c')
```

	Fertility	Agriculture	Examination
Courtelary	80.2	17.0	15
Delemont	83.1	45.1	6
Franches-Mnt	92.5	39.7	5
Moutier	85.8	36.5	12
Neuveville	76.9	43.5	17

Here, we use `booktabs = TRUE` and in the YAML header, as mentioned, we ensured that the `booktabs` LaTeX package is loaded. By loading the `booktabs` package, the typesetting of the tables in LaTeX is often improved over the default options. The `align = 'c'` option ensures that the values in each column in the table are centered. If we want to center the table produced by the above `kable` command, we can use the `kable_styling` function in the `kableExtra` package as follows.

```
library(kableExtra)
swiss_df %>%
  kable(format = 'latex',
        booktabs = TRUE,
        align = 'c') %>%
  kable_styling(position = 'center')
```

	Fertility	Agriculture	Examination
Courtelary	80.2	17.0	15
Delemont	83.1	45.1	6
Franches-Mnt	92.5	39.7	5
Moutier	85.8	36.5	12
Neuveville	76.9	43.5	17

Display maths

In this example, we provide some examples of *display* mode mathematics. As mentioned, this is where the mathematical statements are on separate lines. In this example, we use the `aligned` environment in LaTeX, which allows us to align multiple mathematical statements. Specifically, the example we use is as follows.

```
$$
```

```
\begin{aligned}
y_i &\sim N(\mu_i, \sigma^2),\\
\mu_i &= \beta_0 + \beta_1 x_i + \beta_2 z_i,
\end{aligned}
$$
```

This is rendered as follows.

$$
\begin{aligned}
y_i &\sim N(\mu_i, \sigma^2),\\
\mu_i &= \beta_0 + \beta_1 x_i + \beta_2 z_i,
\end{aligned}
$$

The `&` symbol on each line is used to align the lines, so that the $\sim$ on the first line is aligned with the $=$ on the second line. Here, we need to add `\\` at the end of the first line to force a line break. The `aligned` environment is widely used in LaTeX, especially for showing algebraic derivations. For example, the following

```
$$
\begin{aligned}
y &= \frac{e^x}{1+e^x},\\
  &= \frac{e^x}{e^x (e^{-x} + 1)},\\
  &= \frac{1}{1 + e^{-x}}
\end{aligned}
$$
```

is rendered as follows.

$$
\begin{aligned}
y &= \frac{e^x}{1+e^x},\\
  &= \frac{e^x}{e^x (e^{-x} + 1)},\\
  &= \frac{1}{1 + e^{-x}}.
\end{aligned}
$$

Formatted inline R output

In this RMarkdown example, we display the R^2 of the linear model as well as it accompanying F statistic and p-value with the following code.

```
The $R^2$ for this model is `r round(R_sq, 2)`,
$F(`r round(f_stat[2])`, `r round(f_stat[3])`) =
`r round(f_stat[1],2)`$,
$p `r format.pval(p_value, eps = 0.01)`$.
```

Here, we make use of some quantities, i.e., `R_sq`, `f_stat`, `p_value`, created earlier in the file. In addition, we place some of inline R code inside `$` so that they will be formatted as mathematical notation. Placing inline R code inside either LaTeX mathematical environments is always possible. Consider the following example.

```
$$
\int_{-\infty}^1 \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx = `r round(pnorm(1), 3)`
$$
```

This will produce the following output.

$$
\int_{-\infty}^1 \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx = 0.841
$$

Note that also in this example, we also avail of the R function `format.pval`. This function can be used to return an inequality when the p-value is lower than a certain threshold named `eps`. For example, consider the following example.

```r
p <- c(0.05, 0.02, 0.011, 0.005, 0.001)
format.pval(p, eps = 0.01)
```

```
## [1] "0.050" "0.020" "0.011" "<0.01" "<0.01"
```

Bibliography

In the YAML header, we have included the following statements.

```
bibliography: refs.bib
biblio-style: apalike
```

The first line tells RMarkdown to use the bibliographic information contained in the file `refs.bib`, which is assumed to be present in the working directory. The second line tells RMarkdown to format the citations and the references in the bibliography using APA style.

The `refs.bib` file is just a plain text file, and can be named anything. In this example, its content is minimal and contains just the following content.

```
@book{GelmanHill:2007,
  address = {New York},
  author = {Gelman, Andrew and Hill, Jennifer},
  title = {Data Analysis Using Regression and Multilevel/Hierarchical Models},
  publisher = {Cambridge University Press},
  year = 2007
}
```

The `@book{ .... }` content is a BibTeX bibliographic entry for a book, with BibTeX being the primary bibliography manager used with LaTeX. As this is a simple text file, these bibliographic entries can be relatively easily created. However, because BibTeX is such a widely used bibliographic manager, BibTeX bibliographic entries are provided in many scholarly article data bases, including Google Scholar. It is therefore very easy to build up a personal bibliography file, which can then be re-used in all of one's reports.

Having defined a bibliography in the YAML header as we have done, we can now use the BibTeX *keys* to perform citations. The *key* for BibTeX entry is the text after the first `{` and before the first comma. Consider the following 3 BibTeX entries.

```
@book{xie2017dynamic,
  title={Dynamic Documents with R and knitr},
  author={Xie, Yihui},
  year={2017},
  publisher={Chapman and Hall/CRC}
}

@article{knuth1984literate,
  title={Literate programming},
  author={Knuth, Donald Ervin},
  journal={The Computer Journal},
  volume={27},
  number={2},
  pages={97--111},
  year={1984},
  publisher={Oxford University Press}
}

@book{wickham:2019,
  Author = {Hadley Wickham},
  Title = {Advanced R},
  Publisher = {Chapman and Hall/CRC},
  Year = {2019},
  edition = 2
```

```
}
```

The keys here are `xie2017dynamic`, `knuth1984literate`, and `wickham:2019`. Should we wish to refer to, say, Wickham's book in our RMarkdown file, we would simply write `@wickham:2019`. For example, the following statement in RMarkdown

```
As is described in @wickham:2019, R is a functional programming language.
```

would result in "As in described in Wickham (2019), R is a functional programming language." in the output document. Alternatively, had we written

```
R is a functional programming language [@wickham:2019].
```

this would result in "R is a functional programming language (Wickham 2019)." in the output document.

In either case, the following line would then be inserted at the end of the output document.

Wickham, Hadley. 2019. *Advanced R*. 2nd ed. Chapman; Hall/CRC

If we had multiple citations, they would all be listed at the end of the output document in alphabetical order. By default in RMarkdown, this list of bibliographic references are not given a section name. Therefore, as we have done in this example RMarkdown document, we simply end the document with a section named *References* by using the following line as the last line.

```
# References
```

All the references will now be inserted after this section header. Should we prefer the name *Bibliography*, or another other option here, we simply change the name of this section.

Brief guide to Markdown

As a minimal markup language, there are not too many Markdown commands to learn. Here, we provide an overview of the main ones.

Headers

We have already seen section headers above. Section headers begin with the `#` at the start of a new line followed by the section title. These lines are preceded and followed by a blank line. The number of `#` symbols indicate the level of the section: `#` indicates a section, `##` is a subsection, `###` is a subsubsection, and so on. Note that there should be a space after the `#` symbol or symbols at the start of the line. For example, `# Introduction` will create a section entitled *Introduction*, but `#Introduction` will produce a line with #Introduction on it. In Figure 4, on the left, we see a Markdown file with multiple sections and subsections. On the right, we see this file rendered as a pdf document.

Font style and weight

As we've seen above in a few examples, we can produce italicized text by surrounding the text with `*`, and we obtain bold text by surrounding it with `**`. Here are some examples.

- `*this is italicized text*` produces *this is italicized text*
- `**this is bold text**` produces **this is bold text**
- `***this is bold and italicized text***` produces ***this is bold and italicized text***

Underscores, i.e. `_`, have the same effect as `*`.

- `_this is italicized_` produces *this is italicized*
- `__this is in bold__` produces **this is in bold**
- `___this is in bold and in italics___` produces ***this is in bold and in italics***

We can also mix `_` and `*`.

- `*__this is italicized bold text__*` produces ***this is italicized bold text***

```
# Introduction

This is a sentence in the Introduction.

## Objectives of study

This is a sentence in the first subsection
of the Introduction.

# Analysis

This is a sentence in the Analysis section.

## Exploratory analysis

This is a sentence of the first subsection
of the Analysis section.

## Statistical model

This is a sentence of the second subsection
of the Analysis.

# Conclusion

This is a sentence in the last section
of the document.
```

Introduction

This is a sentence in the Introduction.

Objectives of study

This is a sentence in the first subsection of the Introduction.

Analysis

This is a sentence in the Analysis section.

Exploratory analysis

This is a sentence of the first subsection of the Analysis section.

Statistical model

This is a sentence of the second subsection of the Analysis.

Conclusion

This is a sentence in the last section of the document.

Figure 4: An example of a Markdown file on the left, and its corresponding pdf output on the right. The Markdown file has sections, indicated by lines beginning with `#`, and subsections, indicated by lines beginning with `##`.

- `__*this is italicized bold text*__` produces ***this is italicized bold text***
- `_**this is italicized bold text**_` produces ***this is italicized bold text***

In Markdown, there is no general way of producing underlined text. However, if we are using `pdf_document` output, we can use the LaTeX `\underline` command. For example, `\underline{this text is underlined}` gives this text is underlined.

Code

Often in technical documents, we need to display computer code. If we simply surround the code block by ` ``` `, we will obtain monospace typed text. For example, the following Markdown code shows Python code.

````
```
for x_i in x:
  y.append(x_i)

```
````

This is rendered as follows.

```
for x_i in x:
  y.append(x_i)
```

However, ideally we would prefer the code to be syntax highlighted. We can do so by indicating the code's language after the initial ` ``` `. In the following Markdown code, we state that the code is Python.

````
```python
for x_i in x:
  y.append(x_i)
````
````

````
```
````

This is then rendered as follows.

```
for x_i in x:
  y.append(x_i)
```

When we wish to display R code, we have another option. We can use a normal R code chunk, but set the chunk parameter `eval` to `FALSE`, and set `echo` to `TRUE`, assuming it is not globally set to `TRUE`.

````
```{r, echo=TRUE, eval=FALSE}
n <- 10
x <- rnorm(n)
y <- 2.25 * x + rnorm(n)

M <- lm(y ~ x)
```
````

This is then rendered as follows.

```
n <- 10
x <- rnorm(n)
y <- 2.25 * x + rnorm(n)

M <- lm(y ~ x)
```

Lists

There are three types of lists that are possible with the pdf document output: Itemized lists, enumerated lists, and definition lists.

Itemized lists In itemized lists, also known as unordered lists or even bullet-point lists, each item begins on a new line that begins with `*` followed by a space. The list has to be both preceded and is followed by a blank line. For example, we have Markdown code with an listed list on the left, and its rendered output on the right.

```
* Apple
* Orange
* Blueberry
```

- Apple
- Orange
- Blueberry

Items in a list do not need to be written using only one line of Markdown code. A single item may be spread over multiple lines as in the following example.

```
* If we start our item
on one line,
and continue to another
line with no break,
it will still appear as one item.
* The same thing happens
  if you continue
  on the next line after
  and indention.
```

- If we start our item on one line, and continue to another line with no break, it will still appear as one item.
- The same thing happens if you continue on the next line after and indention.

We can also create nested lists by using indented lines that themselves begin with `*`, as in the following example.
```

```
* Farm animal
    * Cow
    * Pig
    * Sheep
* Wild animal
    * Fox
    * Wolf
```

- Farm animal
    - Cow
    - Pig
    - Sheep
- Wild animal
    - Fox
    - Wolf

In general in Markdown, the indentation should be made with four spaces or a tab. In some cases, fewer spaces than 4 will suffice, but it is generally better to use four spaces. Also, beware that some editors, including the RStudio editor, map the Tab key to 2 spaces.

Enumerated lists Enumerated lists are defined and behave just like itemized lists except that instead of an item or sub-item defined by a line beginning with a `*` (followed by a space), it is defined by a line beginning a number followed by a `.` and then space. In the following example, our items begin with `1.` followed by a space.

```
1. Potato
1. Brocolli
1. Cabbage
```

1. Potato
2. Brocolli
3. Cabbage

We do not need to always use `1.` to obtain an enumerated list. Any other numbers will suffice, as in the following example.

```
1. Potato
4. Brocolli
10. Cabbage
```

1. Potato
2. Brocolli
3. Cabbage

However, if we use a number greater than `1` as the first item, then that will be used as the starting value of the numbering of the list as in the following example.

```
12. Potato
4. Brocolli
10. Cabbage
```

12. Potato
13. Brocolli
14. Cabbage

Enumerated lists can be nested just like itemized lists.

```
1. Tree
    1. Fir
    1. Pine
    1. Oak
1. Flower
    1. Rose
    1. Tulip
    1. Daffodil
```

1. Tree
    1. Fir
    2. Pine
    3. Oak
2. Flower
    1. Rose
    2. Tulip
    3. Daffodil

We may also mix enumerated and itemized lists when using nestings as in the following example.

```
* Tree
    1. Fir
    1. Pine
    1. Oak
* Flower
    1. Rose
    1. Tulip
    1. Daffodil
```

- Tree
    1. Fir
    2. Pine
    3. Oak
- Flower
    1. Rose
    2. Tulip
    3. Daffodil

Definition lists Definition lists, when rendered, begin with an emboldened term followed by a definition or description. This can be useful for definitions per se, but also when we need to elaborate on or describe

certain terms. For example, they could be used to describe the meaning of different variables in a data set. We create a definition list by beginning a new line with some text, which is usually brief, such as a name. The subsequent line then begins with a `:` followed by the definition or description. For example, in the following code, we provide a definition list elaborating upon the variables in a dataset.

```
participant-id
: An integer that uniquely
  codes each participant.

gender
: A binary variable with
  values `female` and `male`.

age
: A numeric variable giving
  the participant's age in years.
```

participant-id An integer that uniquely codes each participant.

gender A binary variable with values `female` and `male`.

age A numeric variable giving the participant's age in years.

Brief guide to mathematical typesetting with LaTeX

RMarkdown provides an extensive set of commands for typesetting mathematical formulas, symbols, and other notation. These commands are in fact LaTeX commands, but are available regardless of the output document format. In other words, even if we are not using the `pdf_document` output format, which is ultimately rendered by a LaTeX typesetting engine, we may still avail of these LaTeX commands for mathematical typesetting.

As we have seen above, there are two *modes* in which mathematical formulas and notation can appear: *inline* mode, and *display* mode. Inline model is where the mathematical notation appears within a line of normal text. It is obtained by surrounding the code a `$`. For example, the following code

```
Einstein's famous formula for mass-energy equivalence is $E = mc^2$.
```

is rendered as follows:

Einstein's famous formula for mass-energy equivalence is $E = mc^2$.

On the other hand, display mode is where the mathematical formulas or notation appear on a line on their own. It is obtained by surrounding the code by `$$`. For example, the following code

```
Einstein's famous formula for mass-energy equivalence is
$$
E = mc^2.
$$
```

is rendered as follows:

Einstein's famous formula for mass-energy equivalence is

$$E = mc^2.$$

In what follows, we provide a brief and minimal introduction to the main mathematical typesetting commands that are available in RMarkdown. To able to appreciate all of the mathematical typesetting options in RMarkdown, we would have to provide a thorough introduction to mathematical typesetting in LaTeX. This is well beyond the scope of this book, but there are many books that provide extensive details about mathematical typesetting in LaTeX. For example, one highly recommended comprehensive and up to date book is Grätzer (2016).

Symbols for variables

In mathematical formulas and notation, there are a large number of symbols that are commonly used as variables. These include the upper and lower case letters of the English alphabet, which can be typed directly.

For example, `$x$` becomes x, `$A$` becomes A, etc. It also very common to use the upper and lower case letters of the Greek alphabet. The lowercase Greek symbols are in the following table.

`\alpha`	α	`\iota`	ι	`\sigma`	σ
`\beta`	β	`\kappa`	κ	`\tau`	τ
`\gamma`	γ	`\lambda`	λ	`\upsilon`	υ
`\delta`	δ	`\mu`	μ	`\phi`	ϕ
`\epsilon`	ϵ	`\nu`	ν	`\chi`	χ
`\zeta`	ζ	`\xi`	ξ	`\psi`	ψ
`\eta`	η	`\pi`	π	`\omega`	ω
`\theta`	θ	`\rho`	ρ		

Given that some uppercase Greek letters are identical to uppercase letters in English, only those that are different to English letters have LaTeX commands or even LaTeX to render them.
These are shown in the following table.

`\Gamma`	Γ	`\Lambda`	Λ	`\Sigma`	Σ	`\Psi`	Ψ
`\Delta`	Δ	`\Xi`	Ξ	`\Upsilon`	Υ	`\Omega`	Ω
`\Theta`	Θ	`\Pi`	Π	`\Phi`	Φ		

Calligraphic or *blackboard bold* fonts of English uppercase letters commonly appear too. The following calligraphic uppercase English letters are often used in mathematical formulas and notation.

`\mathcal{A}`	$\mathcal{A}$	`\mathcal{G}`	$\mathcal{G}$	`\mathcal{M}`	$\mathcal{M}$	`\mathcal{S}`	$\mathcal{S}$	`\mathcal{Y}`	$\mathcal{Y}$
`\mathcal{B}`	$\mathcal{B}$	`\mathcal{H}`	$\mathcal{H}$	`\mathcal{N}`	$\mathcal{N}$	`\mathcal{T}`	$\mathcal{T}$	`\mathcal{Z}`	$\mathcal{Z}$
`\mathcal{C}`	$\mathcal{C}$	`\mathcal{I}`	$\mathcal{I}$	`\mathcal{O}`	$\mathcal{O}$	`\mathcal{U}`	$\mathcal{U}$		
`\mathcal{D}`	$\mathcal{D}$	`\mathcal{J}`	$\mathcal{J}$	`\mathcal{P}`	$\mathcal{P}$	`\mathcal{V}`	$\mathcal{V}$		
`\mathcal{E}`	$\mathcal{E}$	`\mathcal{K}`	$\mathcal{K}$	`\mathcal{Q}`	$\mathcal{Q}$	`\mathcal{W}`	$\mathcal{W}$		
`\mathcal{F}`	$\mathcal{F}$	`\mathcal{L}`	$\mathcal{L}$	`\mathcal{R}`	$\mathcal{R}$	`\mathcal{X}`	$\mathcal{X}$		

Likewise, the following *blackboard bold* fonts are widely used.

`\mathbb{A}`	$\mathbb{A}$	`\mathbb{G}`	$\mathbb{G}$	`\mathbb{M}`	$\mathbb{M}$	`\mathbb{S}`	$\mathbb{S}$	`\mathbb{Y}`	$\mathbb{Y}$
`\mathbb{B}`	$\mathbb{B}$	`\mathbb{H}`	$\mathbb{H}$	`\mathbb{N}`	$\mathbb{N}$	`\mathbb{T}`	$\mathbb{T}$	`\mathbb{Z}`	$\mathbb{Z}$
`\mathbb{C}`	$\mathbb{C}$	`\mathbb{I}`	$\mathbb{I}$	`\mathbb{O}`	$\mathbb{O}$	`\mathbb{U}`	$\mathbb{U}$		
`\mathbb{D}`	$\mathbb{D}$	`\mathbb{J}`	$\mathbb{J}$	`\mathbb{P}`	$\mathbb{P}$	`\mathbb{V}`	$\mathbb{V}$		
`\mathbb{E}`	$\mathbb{E}$	`\mathbb{K}`	$\mathbb{K}$	`\mathbb{Q}`	$\mathbb{Q}$	`\mathbb{W}`	$\mathbb{W}$		
`\mathbb{F}`	$\mathbb{F}$	`\mathbb{L}`	$\mathbb{L}$	`\mathbb{R}`	$\mathbb{R}$	`\mathbb{X}`	$\mathbb{X}$		

As examples of where these fonts are used, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{Q}$ are used to denote the natural numbers, integers, rational, real, and complex numbers, respectively.

Subscripts and superscripts

Subscripts or superscripts are widely used in mathematical notation. For example, all symbols for variables commonly occur with subscripts and superscripts. In addition, subscripts and superscripts occur in mathematical operators and functions, as we will see below. As we've seen in passing in many examples above, we obtain a subscript by using `_` and a superscript using the `^`. For example, `$x_1$` is x_1 and `$x^2$` is x^2. When we need to use more than just a single character or command for the subscript or superscript, we need to surround it with braces. For example, if want 10^{100}, we need to write `$10^{100}`. Writing `$10^100$` will lead to $10^1 00$. Likewise with subscripts. For example, for x_{ijk}, we write `$x_{ijk}$` and not `$x_ijk$`, which would give $x_i jk$.

We may also use braces to nest sub and superscripts as in the following example.

```
2^{2^2}, 2_{i_j}
```

This is rendered as follows:

$$2^{2^2}, 2_{i_j}$$

It should be noted that not using the brace here would simply lead to a `! Double superscript` or `! Double subscript` error.

It is also possible to mix sub and superscripts. For example, `$2^i_j$` gives 2^i_j. Likewise, `$2_j^i$` also gives 2_j^i.

Arithmetic operations and fractions.

For arithmetic, plus and minus are obtained by `+` and `-`, respectively. For example, `$x + y$` gives $x + y$, `$x - y$` gives $x - y$, and so on. The command `\pm` is used to produce the *plus or minus* symbol $\pm$. For exponents, we use superscripts just as we saw above. For multiplication, it is conventional that the absence of any operator implies multiplication. For example, x times y is conventionally often written simply as xy. However, if we prefer to use a symbol to explicitly mark multiplication, we may use `\cdot` or `\times`. For example, `$a \cdot b$` gives $a \cdot b$, and `$a \times b$` gives $a \times b$.

For division, if we are simply dividing one symbol by another, we can use `/` or `\div`. For example, `$a / b$` gives a/b, and `$a \div b$` gives $a \div b$. However, for formulas that involve ratios of set of symbols or larger statements, we need to used `\frac{}{}`, as we see in the following example.

```
$$
\frac{1}{2} + \frac{3}{4} = \frac{4 + 6}{8} = \frac{5}{4}
$$
```

This is rendered as follows:

$$\frac{1}{2} + \frac{3}{4} = \frac{4 + 6}{8} = \frac{5}{4}$$

While it is perhaps more common to use `\frac{}{}` in display mode, it may be used in inline mode as well, as in the following example.

```
The result is $x = \frac{a + b}{c + d}$.
```

This is rendered as follows:

The result is $x = \frac{a + b}{c + d}$.

We may also nest fractions using `\frac{}{}`, as in the following example.

```
$$
\frac{a}{b + \frac{1}{c}}
$$
```

This is rendered as follows:

$$\frac{a}{b + \frac{1}{c}}$$

Sums, products, integrals, etc

Summation over multiple variables is denoted using a variant of the uppercase Sigma symbol. However, for this, we use `\sum` and not `\Sigma`. For example, to denote the sum over a set of numbers $x_1, x_2 \ldots x_n$, we would write this as follows.

```
$$
\sum_{i=1}^n x_i
$$
```

This is rendered as follows:

$$
\sum_{i=1}^{n} x_i
$$

The limits of the sum are not strictly necessary. Thus, we could omit them as follows.

```
$$
\sum x_i
$$
```

This appears as follows:

$$
\sum x_i
$$

However, this summation notation is ambiguous at best, and so including the limits of the sum is highly recommended at all times.

We write products using the uppercase Pi symbol, but again, we should not use `\Pi` but use `\prod` instead. Just like `\sum`, `\prod` should always be used with limits, as in the following example.

```
$$
\prod_{i=1}^n x_i
$$
```

This appears as follows:

$$
\prod_{i=1}^{n} x_i
$$

Note that when sums or products are used in inline mode, they appear in a more compact form. For example, `$\sum_{i=1}^n x_i$` appears as $\sum_{i=1}^n x_i$, and `$\prod_{i=1}^n x_i$` appears as $\prod_{i=1}^n x_i$.

Integrals are written using the `\int` command. Like `\sum` and `\prod`, they may include limits, as in the following example, which gives the area under the curve defined by the function $f(x)$ between the the values of $x = 0$ and $x = 1$.

```
$$
\int_{0}^1 f(x) dx
$$
```

This appears as follows:

$$
\int_{0}^{1} f(x) dx
$$

When limits on the integral are given, it is known as a *definite integral.* In the absence of limits, we have an *indefinite integral* and this is the integral over all values of the variable of integrand.

Roots

We obtain the square root symbol by the `\sqrt` command. To use this properly, unless the expression is a single digit, we must include the expression with the `{}` after the command. In other words, to obtain $\sqrt 2$, we can use `$\sqrt 2$`, but to obtain $\sqrt{42}$, we must write `$\sqrt{42}$`. Had we written `$\sqrt 42$` or even `$\sqrt(42)$`, we would have obtained $\sqrt 42$ or $\sqrt(42)$, which is not the desired result. Note that `\sqrt` produces square root that stretches to enclose the expression to which it applies, as we see in the following example that gives the formula for a sample standard deviation.

```
$$
s = \sqrt{\frac{\sum^n_{i=1} (x_i - \bar{x})^2}{n-1}}
$$
```

This will appear as follows.

$$
s = \sqrt{\frac{\sum^n_{i=1} (x_i - \bar{x})^2}{n-1}}
$$

To obtain the n-th root, we place the value of n in square brackets after `\sqrt` and before the `{}`, as we see in the following example, which displays the cubed root.

```
$$
\sqrt[3]{125} = 5
$$
```

This will appear as follows.

$$
\sqrt[3]{125} = 5
$$

Equalities, inequalities, set operators

Equality and inequalities can be denoted using `=` and `<` or `>`. For example, `$x = y$` gives $x = y$, and `$x < y$` gives $x < y$, and so on. However, there are many other commonly symbols used to denote equalities, equivalences, or inequalities. Likewise, there are many commonly used symbols for set theoretic operations. Some of the more widely used examples are shown in the following table.

`=`	$=$	`\leq`	$\leq$	`\gg`	$\gg$	`\propto`	$\propto$	`\doteq`	$\doteq$	`\subset`	$\subset$	`\supseteq`	$\supseteq$
`<`	$<$	`\geq`	$\geq$	`\approx`	$\approx$	`\equiv`	$\equiv$	`\in`	$\in$	`\supset`	$\supset$	`\cap`	$\cap$
`>`	$>$	`\ll`	$\ll$	`\sim`	$\sim$	`\triangleq`	$\triangleq$	`\ni`	$\ni$	`\subseteq`	$\subseteq$	`\cup`	$\cup$

Many of these operators can be negated using the `\not` command before the operator symbol or command. For example, `$x \not= y$` gives $x \neq y$, `$x \not< y$` gives $x \not< y$, and so.

Multiline and aligned formulas

We may create multiline formulas that are vertically aligned at designated points using `\begin{aligned}`, `\end{aligned}`, as in the following example.

```
$$
\begin{aligned}
f &= (x + y)(x + y),\\
  &= x^2 + 2xy + y^2
\end{aligned}
$$
```

This appears as follows.

$$
\begin{aligned}
f &= (x + y)(x + y),\\
  &= x^2 + 2xy + y^2
\end{aligned}
$$

Note that similarly to the case of the `matrix` environment, we use `&` for alignment and `\\` for newlines.

We may use the `aligned` environment even when we do not need alignment per se, but just require multiline equations, as in the following example.

```
$$
\begin{aligned}
x = a + b\\
y = c + d + e
\end{aligned}
$$
```

This appears as follows.

$$
\begin{aligned}
x = a + b\\
y = c + d + e
\end{aligned}
$$

Git

Version control software (VCS) is an essential tool for the efficient management and organization of source code. In the case of data analysis using R, the relevant source code files will primarily include `.R` and `.Rmd` scripts, but even in small and routine projects, there are many other possibilities too. VCS allows us to keep track of all the versions or revisions to a set of files in an efficient and orderly manner. As a simple example to motivate the use of a VCS system, let us say that we are working on a relatively small data analysis project initially involving some `.R` and `.Rmd` scripts, with names like `preprocessing.R`, `exploration.R`, `analysis.R`, and `report.Rmd`. Let's say that we work with these files, adding new code, editing or deleting old code, etc., every few days in a normal R session. If we were to simply save the files after each session, we would obviously only ever have their most recent versions. All the previous versions would be lost. In order to avoid loss of previous versions in case they are needed, we could periodically *save as*, creating versions like `preprocessing_v1.R`, `analysis_oct23.R`, and so on. As time goes by, it is highly likely too that new files will appear. Some of these may have been only intended to be temporary files, but others might be intended to be vital parts of the project. Some new file might be *branches* of other files, where we copy the original, and work on some new feature of the code in the copy with the intention of merging the changes back if and when necessary. By proceeding in this manner, there is a usually an eventual proliferation of new files and different versions of files with sometimes ambiguous or inscrutable names like `analysis_v1_tmp.R`, `analysis_v1_tmp_new.R`, `preprocessing_tmp_foo.R`, and so on. If files are being copied between different devices or to cloud based storage, and edited on different devices, the situation can get ever more disorganized, with files of similar names but perhaps slightly different contents or different time-stamps across different machines. At this point, especially if we return to this work after a period of time, it is not usually not clear even what each file does, where the latest version of any file is, not to mention what all the previous or temporary versions contain and when and why they were made. If we collaborate with others, things usually become even worse. First, we must decide on a means of sharing files, with email attachments still probably being the default and most widely used method of doing so. Sending back and forth emails with modifications creates yet more versions to manage, and multiple people working independently the same files introduces conflicts that need to be manually resolved. Eventually, we have multiple files and versions, on multiple devices, being edited independently by multiple different people. Knowing what each file and version is or does, and who did what and when and where is usually lost as as a result.

This level of disorganization is frustrating, wasteful of time and effort, and obviously bad for reproducibility. The authors themselves may find it difficult or impossible to pick through their files to recover and reproduce all the steps involved in any analyses. Moreover, even if they were only working on one file such as an RMarkdown file from which their final report was generated, the proliferation of versions across different devices and owners would still occur, making it difficult to pick up and resume their work after a period of inactivity. In addition, they may lose track of which version of the `.Rmd` produced which version of the rendered manuscript. It is all very well knowing that a manuscript was produced by knitting a `.Rmd`, and hence that all its reported results are reproducible in principle, but if we have lost track of the `.Rmd` that produced it, it is obviously no longer reproducible in practice.

VCS systems allow us to manage our source files in an orderly and efficient manner. There are many VCS systems available, both proprietary and open source, and while precise information on usage worldwide is hard

to establish definitively, almost all surveys of VCS usage show that *Git* is now by far the most popular and widely used VCS system. Git is open-source software was originally developed in 2005 for version control of the development of the Linux operating system, something for which it is still used. It gradually became more widely used in the open source community and within a few years had become popular than the previously very widely used open source *subversion* VCS system. With the growing popular of Git hosting sites like GitHub, which currently hosts over 100 million Git based projects, Git is now the most widely used VCS system worldwide.

In what follows, we aim to provide a brief introduction to some of the main features of Git. Obviously, it is beyond the scope of this section to provide a comprehensive introduction to Git. Here, we just provide an introduction to installing and configuring Git, initializing a Git repository, adding files and editing files in the repository, and using a remote repository such as GitHub. These are the *must knows* to get up and running with Git at the start. We do not cover important topics such reverting or resetting changes, branching, merging, rebasing, and so on. This topics, and many others, can be found in books such as Chacon and Straub (2014), which is available in its entirety online at https://git-scm.com/book/en/v2. tory

Installation

Git is available for Windows, MacOs, Linux. Git is first and foremost a command driven software. There are graphical interfaces, i.e. GUIs, to Git, but we will not consider them here and do not recommend them either given that there is only a small number of core commands to learn and they allow Git to be used both efficiently and identically across all different devices.

For Windows, we highly recommend installing and using the *Git Bash* shell available from https://gitforwindows.org/. This provides a Bash Unix shell[1] from which Git can be used just as it would be used on other Unix systems like Linux[2] and MacOs.

For MacOS, Git is already preinstalled on recent versions of MacOS. While this may be perfectly adequate, the preinstalled Git on MacOS is based on a build of Git by Apple and is usually not up to date with the latest version of Git. More recent versions of Git for MacOS are available elsewhere, such as https://git-scm.com/download/mac.

For Linux, given the role of Git in the development of Linux, Git is seen as a vital Linux tool. It is easily installed using the package managers of any Linux distribution, see https://git-scm.com/download/linux.

Once Git is installed, it is available for use using the command `git` in an operating system terminal. For Windows, this means that it is available in the DOS shell using the command `git`. However, if Git is installed, as we recommended, as the Git Bash shell, that Bash shell should be always used instead of the DOS shell. For MacOS and Linux, Git will be available in the system terminal, and will work identically in the Bash, sh, zsh, etc., shells.

For what follows, we will assume users are using a Unix shell, and so all of this will be equally applicable to users of Linux, MacOS, and Windows users assuming they use Git Bash.

To establish that Git has been successfully installed and is available for use, type the following

```
git --version
```

The output of this command on a relatively up to date (as of May, 2021) version could appear as follows.

```
git version 2.25.1
```

Configuration

Before we start using Git, we need to perform some minimal configuration. Specifically, we first need to set our name and email address. Git requires that each time we *commit* to the repository, as we will see below,

[1] https://en.wikipedia.org/wiki/Bash_(Unix_shell)

[2] Strictly speaking, Linux is not Unix, but rather a *nix or Unix-like operating system. However, it is essentially a free and open-source re-implementation of Unix, and so can be seen as Unix for all practical purposes.

we have the name and email address of the person doing the committing. This information could be set on a per commit basis. However, it is more common and easier to set this information as a global configuration setting and then it will be used whenever the user performs a commit. This can be done using `git config` as in the following code.

```
git config --global user.name 'Mark Andrews'
git config --global user.email 'mjandrews.org@gmail.com'
```

It is recommended that we also need to set the text editor that we will use for writing our commit messages, which is also something we will see below. By default, the text editor is the `vi` or `vim` editor. These editors are standard Unix editors. They are loved by some and loathed by others. To the uninitiated, these editors are likely to seen as difficult and probably annoying to use. In any case, it is certainly not necessary to use them as any text editor can be used instead. If a user already has a preference, they should use this. If not, one recommended editor, which is open-source editor and available across the Windows, MacOS, and Linux platforms, is the *Atom* editor, see https://atom.io/. Atom can be set as the default editor to be used with Git as follows.

```
git config --global core.editor "atom --wait"
```

Once set, this configuration information is stored in a *dot file* named `.gitconfig` in the user's home directory. Dot files are a standard Unix file system feature. They are simply files, or possibly even directories, that begin with a dot. They are hidden by default in file listings, and are primarily intended to be used for configuration information. The `.gitconfig` file can be edited at any time to change global configuration settings.

Creating and initiating a Git repository

A Git repository is simply a directory (i.e., a folder) in which all the files, including those in subdirectories, are being tracked and managed, or potentially tracked and managed, by Git. Sometimes, as we will see, we obtain a Git repository by *cloning* it. However, we can always turn any directory on our computer into a Git repository using a single command, as we will see momentarily.

Now let us create a Git repository from a newly created empty directory. There is no necessity for the directory to be newly created and empty, that's just what we use this example. First, let us create a new empty directory `project101` inside of the `Documents` directory in our home directory. We will assume that the directory `Documents` already exists in our home directory, but this is quite common across different platforms. We will use the standard Unix commands `mkdir` to make the `project101` directory, and then use the Unix command `cd` to change directory into it.

```
mkdir ~/Documents/project101
cd ~/Documents/project101
```

We may list the contents of `project101` using the Unix `ls` command. Here, we use `-1aF` option to `ls`. The `-1` asks for the information to be shown with one file or directory per line. The `-a` option will show so-called "hidden" files and directories. Hidden files and directories, also known as *dot* files or directories, begin with .. They are intended to contain configuration information and, by default, not shown in lists of files and directories. The `-F` option is used primarily to indicate whether the items in a directory are files or subdirectories. Directories are listed by ending their name with a '/'.

```
ls -1aF
```

```
./
../
```

From this listing, we see that the directory is empty. The two items that are listed, `./` and `../`, are always present in a listing where we show hidden files. They are merely references to the present (`.`) and parent (`..`) directory.

Now, we create a Git repository in `project101` as follows.

```
git init
```

```
Initialized empty Git repository in /home/rstudio/Documents/project101/.git/
```

From this, we see that Git initializes a repository in `project101`, and created a directory therein named `.git`. We may now look at the contents of `project101` again.

```
ls -1aF
```

```
./
../
.git/
```

We see now that there is a hidden or *dot* directory named `.git`. The presence of the `.git` subdirectory is a necessary and sufficient condition for the directory to be Git repository.

We can now run the `git status` command to see what is the current state of the repository.

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

From the output of this command, we see a number of important pieces of information. First, it tells us that we are on the `master` branch of the repository. What branches are and how to use them is something to which we will return in due course below. Suffice it to say for now that they are a major feature of Git repositories. Next, we see that there are `No commits yet`. This means that we have not yet *committed* anything to the repository and so nothing is being tracked or managed by Git yet. Finally, we also see that there is currently `nothing to commit`. In other words, there are no files in `project101` that could potentially be committed to the repository.

Now, let us put some files into `project101`. These files could be any type of file, but because Git is intended for the management of source code, ideally the files should be text files rather than binary files. For this example, we will put one R script, `script.R`, and one other text file, `readme.md`, into `project101`. These files could be simply moved or copied from some other directory or could be written in an editor, such as RStudio, and then saved into `project101`.

For this example, we will assume that the contents of `script.R` is the following.

```
library(tidyverse)

survivors <- Titanic %>%
  apply(c('Sex', 'Survived'), sum)
```

We will assume that the contents of `readme.md` is as follows.

```
The project contains an R script, `script.R`, for processing the `Titanic` data set.
```

After we have put these files in `project101`, we can do a file listing as we did above.

```
ls -1aF
```

```
./
../
.git/
readme.md
```

```
script.R
```

We see that `script.R` and `readme.md` are there. Next, we can do a `git status` as above.

```
git status
```

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    readme.md
    script.R

nothing added to commit but untracked files present (use "git add" to track)
```

Much of the information here is as it was previously, but now we are told that there are two `Untracked files`, namely `readme.md` and `script.R`. Files listed as *untracked* mean that they are present in the directory but not as yet being tracked or managed by Git. As the output indicates, however, we can use the command `git add` to get Git to track them, as we do in the following command.

```
git add readme.md script.R
```

Now, let us check the status of the repository again.

```
git status
```

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   readme.md
    new file:   script.R
```

We see from the output that there are new files, `readme.md` and `script.R`, that *can* be committed. This is an important point. The files are not yet committed to the repository, they are *staged* for commitment. When files are staged, they are in an intermediate area, kind of like a departure lounge in an airport. To commit them, we must run the `git commit` command as follows.

This will open your editor, and it will contain the following text exactly.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   readme.md
#   new file:   script.R
#
```

In other words, the lines beginning with `#` provide information to you as you write your commit message, but they will not be part of the message itself. You write your message above these `#` lines. It is conventional

and recommended[3] that the first line of this message is no more than 50 characters long and is followed by a blank line and then followed by more elaboration. The first line is treated as a subject line. Its first character should be capitalized, and it should not end in a full-stop/period. It is also recommended that this subject line be written in imperative tense and not past tense. For example, it is recommended that we write something like "Add new function ..." rather than "Added new function ...". Admittedly, this is probably initially an unnatural way to write for most people. After the subject line, there must be a blank line. Without it, some features of Git can be affected. Then, a more elaborate message can be written. Here, the imperative tense is no longer necessary. In fact, it is not necessary to have a body at all, but it is highly recommended to provide a message body and to use it provide details about what the code being committed does. It will be helpful for others, including your future self, to understand what was being added and why. The character line of the body text is recommended to to 72. Most Git-aware editors, like Atom, will indicate if the subject line is over 50 characters, and if there is not a blank line after the subject, and will wrap the body at 72 characters.

In this example, because this commit is our first commit, our subject line should acknowledge that this is the beginning of the project. As such, a message like the following can be used.

```
Initialize the repository

Two files are added.
  * `script.R` is an R script summarizing the `Titanic` data-set
  * `readme.md` is the project's readme file.
```

Let us assume, therefore, that we this is what we have typed into the editor after we ran the `git commit` command the editor was opened. After we save this file and quit the editor, the commit is completed.

Before proceeding, we should note that it is also possible to write one-liner commit messages using the option `-m` as follows.

```
git commit -m 'Do something to something'
```

Here, no editor opens and there's no body to the message. For simplicity, we will use this method below sometimes. However, using this method is probably to be avoided as it will, by necessity, lead to minimal and probably poorly thought out commit messages.

We can now view at the Git log of the repository with the following command.

```
git log
```

The output of `git log` would be something like this.

```
commit 59bb2c5ed72bc96a1f791f0f1eae4eb56caa1a1a
Author: Mark Andrews <mjandrews.org@gmail.com>
Date:   Sun May 2 13:39:32 2021 +0000

    Initialize the repository

    Two files are added.
      * `script.R` is an R script summarizing the `Titanic` data-set
      * `readme.md` is the project's readme file.
```

As we can see, our commit message is there, as is our name, email address, and the date and time stamp of the commit. A crucial additional piece of information is the *commit hash*, which in this case is

```
59bb2c5ed72bc96a1f791f0f1eae4eb56caa1a1a
```

The commit hash is a vital feature of Git. It is a 40 hexadecimal character (160 bit) cryptographic hash of the contents and other defining information about each commit. In other words, it can be seen as essentially

[3] See https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

a fingerprint of the commit, but not just an arbitrarily assigned identifier, but one that is calculated using a hashing algorithmic from the contents of the commit. It can therefore be used to uniquely identify the commit and to do a file integrity check of its contents.

Having performed the commit, we can now again check the status of the repository.

```
git status
```

```
On branch master
nothing to commit, working tree clean
```

As we can see, now the repository is in a clean state. All the original files have been committed. There are as yet no new files in the directory, and no edits to the existing files yet either.

As an interim summary, thus far, we have seen a number of essential and regularly used Git commands.

`git init` Create a Git repository in the current working directory of the shell.

`git status` Report the current working status of the repository. Specifically, are there untracked files in the repository or files in the staging area that have not yet been committed.q

`git add <file> ...` Add the files `<file> ...` to the staging area. The staging area is like the departure lounge of an airport. The files therein are going scheduled for committal, though they may be taken out of the staging area too.

`git commit` Commit the files in the staging area. This command opens an editor and a commit message is entered there.

Adding and editing files

As with any data analysis project, as it progresses, new files will be made and edits will be made to existing one. With Git, we can choose whether and when to add new files to the repository. In other words, for example, if new files are added to the directory `project101`, they will be treated as *untracked* files. They will never be automatically added to the repository. We must explicitly add them, using `git add`. Moreover, as we've seen above, adding files with `git add` only puts them in the staging area for committal. They are not committed until we explicitly commit them with `git commit`. Something similar occurs with edits to the existing files. After any edits, Git identifies that files have be modified. However, for these changes to be committed to the repository, they must first be added to the staging area with the `git add` command. Then, they must be explicitly committed with a `git commit` command. This double step process allows us to build up the staging area gradually as we work, and then committing all its contents when we they are all ready. This intended purpose of this is that we can then commit a set of files and edits that are all related to one another and together effectively do one main thing, such as fix a bug or a new feature, etc. These are known as *atomic commits*.

```
cat << "EOF" > models.R
model <- lm(Fertility ~ Catholic, data = swiss)
model_summary <- summary(model)
```

```
bash: line 2: warning: here-document at line 0 delimited by end-of-file (wanted `EOF')
```

Continuing with our very simple example project, let us now add a new file named `models.R` whose contents is as follows.

```
model <- lm(Fertility ~ Catholic, data = swiss)
model_summary <- summary(model)
```

In addition, let us edit the `script.R` file by changing the function `sum` for `mean` in the `apply` function.

Now, let us check the working state of the repository.

```
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   script.R

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    models.R

no changes added to commit (use "git add" and/or "git commit -a")
```

From this, we see that there is, as expected, one untracked file, `models.R`, and one modified file, `script.R`.

We may now add both of these files, and the commit them both to the repository at the same time using a single commit. Alternatively, we could add and commit them individually. Whether we proceed one way or another should be based on whether the commit is atomic, i.e. has one main unitary function or purpose. In this example, because the new file and the change to `script.R` are not related to one another, we will perform two separate commits. First, we will add the modified `script.R` to staging.

```
git add script.R
```

Before we proceed, let us check the working state of the repository.

```
git status
```

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   script.R

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    models.R
```

As we can see, `script.R` is now in the staging area ready to be committed. We now can do the commit.

```
git commit
```

This will bring up our editor with the following contents.

```

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#   modified:   script.R
#
# Untracked files:
#   models.R
#
```

As before, we should write our commit message above the lines beginning with `#`, following the conventions and recommendations for good git messages. We will add the following text to the commit message in the editor.

```
Change function used in the apply functional
```

```
The functional now calculates the mean number of men and
women who survived the Titanic, not the total number.
```

Then, as before, when we save and quit, the staged changes are committed. If we look at the logs, we will now see the following.

```
git log
```

```
commit 38dfdc172fd1f7eb17c7651e6c0bee62ff79b204
Author: Mark Andrews <mjandrews.org@gmail.com>
Date:   Sun May 2 14:39:32 2021 +0000

    Change function used in the apply functional

    The functional now calculates the mean number of men and
    women who survived the Titanic, not the total number.

commit 59bb2c5ed72bc96a1f791f0f1eae4eb56caa1a1a
Author: Mark Andrews <mjandrews.org@gmail.com>
Date:   Sun May 2 13:39:32 2021 +0000

    Initialize the repository

    Two files are added.
      * `script.R` is an R script summarizing the `Titanic` data-set
      * `readme.md` is the project's readme file.
```

Let us again check the repository's status.

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    models.R

nothing added to commit but untracked files present (use "git add" to track)
```

As expected, we now see that there is just one untracked file, `models.R`, in the directory. We first add this file to put it in the staging area.

```
git add models.R
```

Then, we commit it with the following message.

```
Add new lm script named `models.R`

This script performs a linear regression analysis of the
built-in `swiss` data-set.
```

The status of the repository should confirm that everything is now clean.

```
git status
```

```
On branch master
nothing to commit, working tree clean
```

We can also check the logs.

```
git log
```

```
commit b8b1c6ce14fcdd22a4c91c916e62c2c64181b15c
```

```
Author: Mark Andrews <mjandrews.org@gmail.com>
Date:   Sun May 2 14:39:32 2021 +0000

    Add new lm script named `models.R`

    This script performs a linear regression analysis of the
    built-in `swiss` data-set.

commit 38dfdc172fd1f7eb17c7651e6c0bee62ff79b204
Author: Mark Andrews <mjandrews.org@gmail.com>
Date:   Sun May 2 14:39:32 2021 +0000

    Change function used in the apply functional

    The functional now calculates the mean number of men and
    women who survived the Titanic, not the total number.

commit 59bb2c5ed72bc96a1f791f0f1eae4eb56caa1a1a
Author: Mark Andrews <mjandrews.org@gmail.com>
Date:   Sun May 2 13:39:32 2021 +0000

    Initialize the repository

    Two files are added.
      * `script.R` is an R script summarizing the `Titanic` data-set
      * `readme.md` is the project's readme file.
```

As expected, we now have three commits in the log now.

Using remote repositories

Thus far, we have been using Git locally on one computer, and there has been only one user. However, one of the key reasons for using Git, or any other VCS system, is for sharing and collaborating on projects. For this, we need to use *remote* repositories. It is both inexpensive, or possibly even free, and not technically difficult to host your own private Git server, which can then be used for either private team work or for sharing projects with the public. The required software, including the operating system of the server, i.e. Linux, is open-source and free, and so the only expense is hiring a server from a hosting company[4] However, we will not consider this option further here. Instead, we will consider special purpose Git hosting sites, particularly GitHub. GitHub is extremely popular, with around over 40 million users and 100 million repositories as of early 2020.

To share your Git repository, the first step is to create a new repository on GitHub. This assumes that you already have a GitHub account. GitHub provides free (no cost) and paid-for accounts. For most purposes, the free accounts are more than sufficient.

Assuming we have a GitHub account and have logged in, we then can browse to the following url.

`https://github.com/new`

This will bring up a web page that will allow us to create a new empty repository. See Figure 5 for a screenshot of this page. When asked for the repository name, you can use any name. In Figure 5, we use `project101`, but it is not necessary to use the same name for the remote and local repository. The contents of the repository will define it, not its name. You can then add a description. This is just something for the GitHub listing itself. This description will not part of the repository in general. Then, assuming you've already created a local repository that you wish to push to GitHub, leave everything else on this page at its

[4]From companies such as https://www.digitalocean.com/, this may be between $5-$10 USD per month.

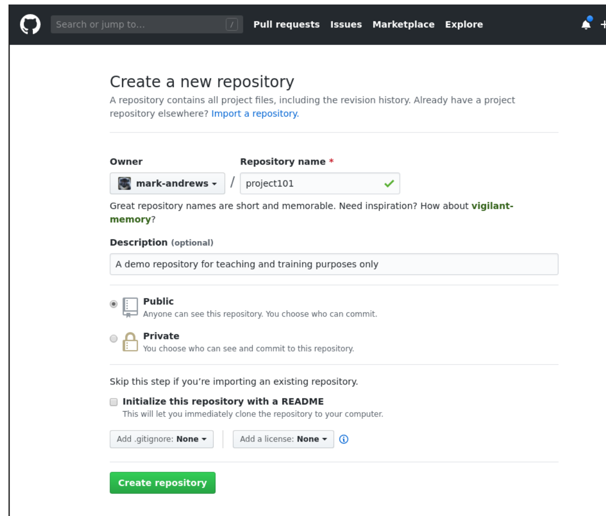


Figure 5: A screenshot of a GitHub page for creating a new repository.

default. Specifically, do *not* add a `readme`, or a licence, or a `.gitignore` file. As important as these are, we can add them later using Git commands.

After we click the "Create repository", we are then brought to a page that provides some Git commands for different situations. We already have an existing repository, so we want the code listed under the heading *. . . or push an existing repository from the command line*, which is

```
git remote add origin git@github.com:mark-andrews/project101.git
git push -u origin master
```

There are two important commands here, and so we will look at them individually. The first command adds `github.com:mark-andrews/project101.git` as a remote host of the repository we have created. In this repository, this remote repository will be named `origin`. The name `origin` is the default name for remote repositories from which other repositories are *cloned*. However, we do not have to use this name, and in fact another name, such as `github`, might be more useful, especially if we have multiple remote repositories. The second command *pushes* the contents of the master branch of local repository to the remote repository named `origin`. When you run this second command, you will be asked for your GitHub password, unless you have set up passwordless GitHub authentication using ssh keys, which is a very convenient feature when frequently using GitHub.

Remember that we must run these command in the Git shell in the, in this example, `project101` repository that we have been using above. After we run them, if we then browse in a web browser to https://github.com/mark-andrews/project101 we will see that our repository and all its history and other vital information is being now hosted there.

Cloning remotes

If you create a public GitHub repository, as we did above, anyone can now *clone* your repository. For example, in a MacOS or Linux terminal or the Windows Git Bash shell, anyone can type the following command and then clone `project101`.

```
git clone git@github.com:mark-andrews/project101.git
```

Alternatively, they could use this version of the clone command.

```
git clone https://github.com/mark-andrews/project101.git
```

The difference between these two versions is simply the internet protocol that is being used. Having cloned

`project101`, they will have access to everything that you pushed to GitHub, i.e. the master branch and all the history and other vital features of the repository. They will be able to do everything you can do with the repository: view the logs, make changes, commit changes, undo changes, roll back history, etc. All the logs, commit messages, commit hashes etc will be identical in the clone as in the original. However, as is probably obvious, whatever actions they take, they will not be able to affect your local repository in any way. Moreover, they will not be able to affect your remote repository on GitHub either. They would only be able to `push` to your GitHub repository if they had a GitHub account and you explicitly gave them `push` permission.

References

Chacon, Scott, and Ben Straub. 2014. *Pro Git.* 2nd ed. Apress.

Fecher, Benedikt, Sascha Friesike, and Marcel Hebing. 2015. "What Drives Academic Data Sharing?" *PloS One* 10 (2): e0118053.

Gorgolewski, Krzysztof J, and Russell A Poldrack. 2016. "A Practical Guide for Improving Transparency and Reproducibility in Neuroimaging Research." *PLoS Biology* 14 (7): e1002506.

Grätzer, George. 2016. *Math into Latex.* 5th ed. Springer Science & Business Media.

Gruber, John. 2004. "Markdown." https://daringfireball.net/projects/markdown/.

Hern, Alex. 2013. "Is Excel the Most Dangerous Piece of Software in the World?" *New Statesman.* https://www.newstatesman.com/technology/2013/02/excel-most-dangerous-piece-software-world.

Houtkoop, Bobby Lee, Chris Chambers, Malcolm Macleod, Dorothy VM Bishop, Thomas E Nichols, and Eric-Jan Wagenmakers. 2018. "Data Sharing in Psychology: A Survey on Barriers and Preconditions." *Advances in Methods and Practices in Psychological Science* 1 (1): 70–85.

Ioannidis, John PA. 2015. "How to Make More Published Research True." *Revista Cubana de Información En Ciencias de La Salud (ACIMED)* 26 (2): 187–200.

Iqbal, Shareen A, Joshua D Wallach, Muin J Khoury, Sheri D Schully, and John PA Ioannidis. 2016. "Reproducible Research Practices and Transparency Across the Biomedical Literature." *PLoS Biology* 14 (1): e1002333.

Knuth, Donald Ervin. 1984. "Literate Programming." *The Computer Journal* 27 (2): 97–111.

Landau, William Michael. 2018. "The Drake R Package: A Pipeline Toolkit for Reproducibility and High-Performance Computing." *Journal of Open Source Software* 3 (21). https://doi.org/10.21105/joss.00550.

MacFarlane, John. 2006. "Pandoc: A Universal Document Converter." https://pandoc.org/.

Merton, Robert K. 1973. *The Sociology of Science: Theoretical and Empirical Investigations.* University of Chicago press.

Munafò, Marcus R, Brian A Nosek, Dorothy VM Bishop, Katherine S Button, Christopher D Chambers, Nathalie Percie Du Sert, Uri Simonsohn, Eric-Jan Wagenmakers, Jennifer J Ware, and John PA Ioannidis. 2017. "A Manifesto for Reproducible Science." *Nature Human Behaviour* 1 (1): 0021.

Nosek, Brian A, George Alter, George C Banks, Denny Borsboom, Sara D Bowman, Steven J Breckler, Stuart Buck, et al. 2015. "Promoting an Open Research Culture." *Science* 348 (6242): 1422–5.

Shamir, Lior, John F Wallin, Alice Allen, Bruce Berriman, Peter Teuben, Robert J Nemiroff, Jessica Mink, Robert J Hanisch, and Kimberly DuPrie. 2013. "Practices in Source Code Sharing in Astrophysics." *Astronomy and Computing* 1: 54–58.

Stodden, Victoria, Peixuan Guo, and Zhaokun Ma. 2013. "Toward Reproducible Computational Research: An Empirical Analysis of Data and Code Policy Adoption by Journals." *PloS One* 8 (6): e67111.

Tenopir, Carol, Suzie Allard, Kimberly Douglass, Arsev Umur Aydinoglu, Lei Wu, Eleanor Read, Maribeth Manoff, and Mike Frame. 2011. "Data Sharing by Scientists: Practices and Perceptions." *PloS One* 6 (6): e21101.

Xie, Yihui. 2017. *Dynamic Documents with R and Knitr*. Chapman; Hall/CRC.