Introducing Markdown and Pandoc

Using Markup Language and Document Converter

Thomas Mailund

Apress®

Introducing Markdown and Pandoc

Using Markup Language and Document Converter

Thomas Mailund

Apress®

Introducing Markdown and Pandoc: Using Markup Language and Document Converter

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-5148-5 ISBN-13 (electronic): 978-1-4842-5149-2
https://doi.org/10.1007/978-1-4842-5149-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484251485. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. He has a background in math and computer science. For the past decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species. He has published *R Data Science Quick Reference*, *The Joys of Hashing*, *Domain-Specific Languages in R*, *Beginning Data Science in R*, *Functional Programming in R*, and *Metaprogramming in R*, all from Apress, as well as other books.

About the Technical Reviewer

Germán González-Morris is a polyglot software architect/engineer with 20+ years in the field, with knowledge in Java(EE), Spring, Haskell, C, Python, and Javascript, among others. He works with web distributed applications. Germán loves math puzzles (including reading Knuth) and swimming. He has tech-reviewed several books, including an application container book (Weblogic), as well as titles covering various programming languages (Haskell, Typescript, WebAssembly, Math for coders, and regexp). You can find more details at his blog site (`https://devwebcl.blogspot.com/`) or twitter account (`@devwebcl`).

CHAPTER 1

The Beginner's Guide to Markdown and Pandoc

Markdown is a markup language. The name is a pun, but where the humor might be atrocious, the language is not. The Markdown language lets you write plain text documents with a few lightweight annotations that specify how you want the document formatted. Such annotations are the defining characteristics of a markup language. Markup languages separate the semantic or content part of a document from the formatting of said document. The content of a document is the text, what should be headers, what should be emphasized, and so on. The formatting specifies the font and font size, whether headers should be numbered, and so on.

Markup languages have a stronger focus on semantic information than direct formatting as you would do with WYSIWYG (what you see is what you get) formatting. With markup languages, you might annotate your text with information about where chapters and sections start, but not how chapter and heading captions should be formatted. Decoupling the structure of a text from how it is visualized makes it easier for you to produce different kinds of output. The same text can easily be transformed into HTML, PDF, or Word documents by tools that understand the markup annotations. And because writing the text and formatting it are

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_1

separate steps, you can apply one or more text documents to the same transformation program to get a consistent look for related documents, or you can transform the same document into multiple output formats so the same document can be put on a web page or in a printed book, for example. Most WYSIWYG editors can export to different formats, but they usually do not let you output to the same document type with different formatting, for example, output PDF files in A4, 6″ x 9″, and 7″ x 10″ with point size 11 in the first two and 12 in the last. With a Markup language, this is relatively easy.

Among markup languages, Markdown is one of, if not the, simplest. The annotations you add to a text are minimal, and most likely you will already have seen most of them if you occasionally use plain text files. For example, where you would use italic or boldface in Word, you would write ∗italic∗ and ∗∗boldface∗∗ in Markdown, and most likely you have seen this notation before. In my misspelled youth, I frequently used TeX/LaTeX and HTM-L/SGML/XML. I know people who cannot concentrate on the text body if it is full of markup information. With Markdown, the markup annotation is almost invisible, and they have no problem working with that. With Markdown you can generate documents in other markup languages, so you do not need to know them. If you want the full power to format your documents the way you want them, then I still recommend that you learn the other languages. You can use that knowledge to create templates (see Chapter 9), and then you only need to use, for example, LaTeX or HTML when writing the templates. You can then still keep your document in Markdown. One exception, where you still want to use LaTeX, is if you need to write math in your document. Then you need to write it in LaTeX; see Chapter 6.

You need a program for translating Markdown into other file formats. The tool I will use in this book is Pandoc. Pandoc supports basic Markdown and several different extensions. It also lets you define templates and stylesheets to customize the transformed files. Pandoc can

do more than translate Markdown files into different output files. It can translate from and to several different formats. I will only describe how you translate from Markdown to other formats. If you have an existing text in Word, for example, and you want to try out Markdown by editing that document, then you should be able to generate a Markdown file from the Word file, edit the Markdown format, and then translate the Markdown document back to Word.

CHAPTER 2

Why Use Markdown and Pandoc

If you are used to WYSIWYG editors such as Microsoft Word, you might reasonably ask why you should use Markdown files. You can write your document and format them any way you like, and you can export your document to different file formats if you wish. For short documents that you only need to format once and to one file format, you do not need Markdown. I will argue that Markdown is still an excellent choice for such documents, but it is for more advanced applications where it really shines.

For applications that are just as easy to handle with a WYSIWYG editor, plain text can be a better choice in situations where you need to share documents with others. A de facto file format for this is Word files, but not everyone has Word. I don't. I can import Word files into Pages, which I have, and export to Word, but I don't know what that does to the formatting. Everyone has an editor that can work on plain files, and with a plain text file, you know exactly what you are editing. If the text and the formatting are separated, then someone with more artistic skills can handle the formatting while I can write the text. One argument for Word might be tracking of changes. This is an important feature, but with plain text files, you can put them under real version control, for example, GitHub, and that is superior to version tracking.

If you need your document in different formats, for example, you might need to include your text in a printed progress report and also have it on a

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_2

web site, then you can export the document to as many file formats as you need. If you need different typography for the different file formats, you might have to do substantial manual work. You might need to change all the document styles by hand, and in the numerous occasions where you need to make changes to your text, you need to change the styles for each file format more than once. If you separate style and text, you avoid this problem altogether.

Using a markup language to annotate your text makes it easier for you to distinguish between the semantic structure of a document and how it is formatted. In the Markdown document, you markup where headers and lists are, for example, but not how these should be formatted in the final output. The formatting styles are held in different files and you can easily transform your Markdown input into all the output file formats and styles you need. Furthermore, someone else can work on the style specification while you concentrate on the text. Your Markdown doesn't have to be in a single file either. You can split it into as many as you want, and then different authors can work on separate pieces of the text without worrying about how to merge files afterward. With version control, you can even work on the same file in parallel up to a point.

Separating Semantics from Formatting

Most documents have a semantic structure. Texts consist of chapters and sections, plain text and emphasized text, figures and citations, quotes, and lists. When we read a document, these semantic elements are visualized by different fonts, bold and italic text, different font sizes, and we do not directly see the semantic structure. Because we don't immediately see the structure, it is easy to forget that it is there.

Most word processors separate semantics from formatting. If you take care to use the formatting section when working on a Word document, then the semantic information needed to change styles, that is, the visual

representation of all semantic units (e.g., headings) is readily available. Separating the semantics of a document from its formatting is not an exclusive property of markup languages. However, when the separation of text and semantics is not enforced, there is a potential for error. If you decide to change the font size of level-two section headers, for example, you can easily do this, but you can equally easy highlight a single section header and reformat that, changing only that single header. That makes this particular header different from all the rest, and if you later modify the formatting of level-two headers, you won't be changing this one header. Great if this is on purpose; not great if this is not what you wanted.

With WYSIWYG editors, you can separate semantics from formatting, but it is easy to break this separation. With markup languages, you can also define some text elements as special and their format different from related items, but you have to do this explicitly so you cannot easily do this by mistake. Keeping the core text consisting of semantic elements and separate from formatting is vital in many situations. If you want to translate your text into both paper documents and web pages, you typically want the format to be different in the two resulting documents. If the core text only contains the semantic structure, this is quickly done, by having a different mapping from semantic elements to formatting information, typically called *templates* or *stylesheets* (see Chapter 9). With different stylesheets for different output formats, the formatting is tied to the output text rather than the input text (see Figure 2-1).

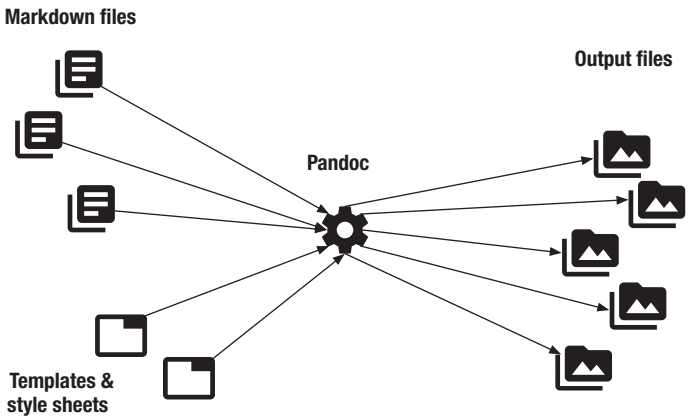

Figure 2-1. *You can translate the text in multiple documents, or multiple chapters that should be merged into a single document. You combine these with templates for formatting the documents, and using Pandoc you can combine it all to produce the documents you want.*

Explicitly representing the semantic elements in the text, rather than implicitly through how the text is formatted, is also essential if you want to automatically make a table of contents or lists of figures and tables. If all sections are marked up explicitly as sections, with headers at different section levels, any tool can scan your document and identify these. If the tools had to guess at the semantic meaning of text elements, based on how the text was formatted, this would be a much harder task.

Using WYSIWYG word processors doesn't prevent you from structuring your documents as semantic units—they usually support this—but having an explicit markup language makes it much easier to enforce.

Preprocessing Documents

If your documents are in plain text, you also get a lot of options for how to process your document before you format it into a final output. There are a large number of tools that will work well with plain text and let you preprocess your documents.

Preprocessing documents often require a few programming skills, so it might not be the first thing you want to worry about if you are only

interested in writing text, but since the option is there, you can write your text without worrying about processing it initially, and add such steps later.

I write a lot about R programming, and in those books, I have a lot of code examples. Here, I use another preprocessor, one that lets me evaluate the code when processing the documents so I know that all the code examples work and so I can get the output of running code inserted into the documents automatically before I create the output formats.

Preprocessing your documents adds some complications to how you format your text, but the complexities are only there when you need them. If you do not need a preprocessor, then you can ignore that they exist altogether. If you *do* need preprocessing, then read Chapter 10.

Why Markdown?

There are many different markup languages you can use. HTML (hypertext markup language) is used for web pages. TeX and LaTeX are used for many kinds of text documents but are especially powerful for typesetting mathematics. Markdown is what we do on in this book.

What makes Markdown particularly pleasant to work with is its simplicity. In HTML, for example, you need to structure your text using tags that enclose every paragraph, every header, every list, and so on. When you edit an HTML document, it is hard to separate the annotations from just the text you want to write. LaTeX has the same problem. The annotation of the text can be hard to ignore when you want to focus on writing.

Worse, if you write your documents in HTML or LaTeX, much of the text is markup codes that specify the formatting. How much, of course, depends on your document, but any markup instructions you make can make the text difficult to read.

Consider this Markdown document:

```
# This is a level one header

This is a paragraph
```

```
## This is a level two header

Here is a paragraph that is followed by

  * an unnumbered
  * list

1. and a numbered
2. list that is
3. three items long
```

I hope you will agree that the markups here are minimal and that they do not get in the way of reading or writing the text.

For comparison, the HTML version of the same text looks like this:

```
<h1>This is a level one header</h1>
<p>This is a paragraph</p>
<h2>This is a level two header</h2>
<p>Here is a paragraph that is followed by</p>
<ul>
  <li>an unnumbered</li>
  <li>list</li>
</ul>
<ol type="1">
  <li>and a numbered</li>
  <li>list that is</li>
  <li>three items long</li>
</ol>
```

It is not terribly complicated, and after looking at it a bit, you can certainly follow the structure of a document. It is far from as clean as the Markdown file.

The LaTeX version is slightly easier to read than the HTML file, but there are still several formatting instructions that get in the way of just writing.

```
\section{This is a level one header}
This is a paragraph
```

```
\subsection{This is a level two header}
Here is a paragraph that is followed by

\begin{itemize}
\item an unnumbered
\item list
\end{itemize}

\begin{enumerate}
\item and a numbered
\item list that is
\item three items long
\end{enumerate}
```

Markdown is designed so you can annotate your text with semantic information with little annotation clutter. It is designed such that reading the input text is almost as easy as reading the formatted text. With Markdown you don't have quite the same power to control your formatting as you do in a language like LaTeX, but the simplicity of Markdown more than makes up for it.

Why Pandoc?

Since Markdown is just a language for adding structure to a text, it is not tied to any particular tool. You can use any Markdown-aware software when you want to process your documents. Many blogging platforms will let you write your text in Markdown and automatically format it for you. Translating Markdown into HTML was, after all, one of the primary motivations for the language. Now, many text editors also support Markdown and will support formatting in Markdown and exporting to various file formats, usually with various formatting and style choices determining what your output files will look like.

If your editor can export to different file formats and in different styles, then that is obviously the easiest way for you to export your Markdown text. With Pandoc, however, you have a lot of power over how your documents should be processed. Pandoc is vastly more versatile than any Markdown-aware text editor that I am aware of.

If you want to create a simple document with no fluff, it is easy to do so with Pandoc, but easier to do from inside your editor. Try using Pandoc for simple cases though, so you get familiar with the tool. When you get into serious writing, and you want full control of how your final documents will look, then you need the power of Pandoc. The learning curve can be steep, but if you are familiar with using Pandoc for simple documents, then you have a foundation to build on when you explore advanced features.

CHAPTER 3

Writing Markdown

If you have used plain text to write and share documents in the past, then you are likely to be familiar with most Markdown markup annotations already. Much of the syntax for Markdown is based on how people have written plain text documents for years. This chapter covers the basic Markdown annotations, which make up 99% of the annotations you will use regularly. The notation I present in this chapter is supported by all Markdown tools (as far as I am aware). The next chapter covers notation that is not universally supported although many tools do support the features there. All of them, of course, are supported by Pandoc.

If you write plain text with no special markup commands, as listed in the remainder of this chapter, then the result will be plain text in the output as well. One or more lines of text becomes a paragraph. If you need to start a new paragraph, then use two new lines, that is, separate one paragraph from the next with a blank line.

Sections

At the highest level, a text document is composed of its sections. Sections come at different levels. In a book, the top level might be chapters, the second level sections within the chapters, and the third level are subsections within the sections.

To make a new section, you give it a header. The headers start with a hashtag. Using one hashtag gives you a level-one header, which will

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_3

be a chapter in a book or a section in a smaller document. Two hashtags give you a level-two section, a section if the first level is chapters or a subsection if the first level is section. The next level sections have three hashtags, and so on.

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

For the first two levels, you can alternatively underline the section titles with = and -, respectively:

```
Level  one header
=================
Level  two header
-----------------
```

Any text you write following a header becomes the body of the section.

By default the headers are numbered. You can change this using a template (see Chapter 9) or you can disable numbering on selected headers by putting "{-}" or "{.unnumbered}" after the header title:

```
# Unnumbered header {-}
## Another unnumbered header { .unnumbered }
```

Emphasis

We emphasize part of a text by putting it in italic or boldface. In Markdown we use asterisks, *, to do this. To put a word in italic, we use *one* asterisk, and to put it in boldface, we use **two**.

The preceding section, in Markdown, looks like this:

```
We emphasise part of a text by putting
it in italic or boldface. In Markdown
we use asterisks, \*, to do this.
```

```
To put a word in italic, we use *one*
asterisk and to put it in boldface we
use **two**.
```

You will also notice that the first asterisk is escaped using a backslash. That backslash prevents Markdown from interpreting the asterisks as the beginning of an italic text.

Lists

You have two kinds of lists: numbered and unnumbered. To create a numbered list, you put a number, followed by a period, at the start of a line and write the list item after it. For the next list item, you go to the next line, add another number, followed by a dot, and write the next item text. An example could look like this:

```
1. This is a numbered list.
2. Where this is list item two.
3. And this is list item three.
```

The result will look like this:

1. This is a numbered list.
2. Where this is list item two.
3. And this is list item three.

The actual numbers are ignored, so you can get the same result if you wrote:

```
10. This is a numbered list.
51. Where this is list item two.
42. And this is list item three.
```

If you want your list items to span multiple lines, you need to indent the lines following the number, like this:

```
1. This is a multi-line list item.
   This is also part of the list item.
   And so is this
2. Here is another one.
   Where this is also part of the list item.
```

Doing that will give you this list:

1. This is a multi-line list item. This is also part of the list item. And so is this.
2. Here is another one. Where this is also part of the list item.

For unnumbered lists, you use an asterisk or a dash instead of numbers; so you can write an unnumbered list like this:

```
* This is a numbered list
- Where this is list item two
* And this is list item three
```

- This is a numbered list
- Where this is list item two
- And this is list item three

As you can see, you can mix asterisks and dashes, or stick to any of the two you prefer. How the list is formatted when you create a document is determined by the stylesheet and not which symbol you use to create the list.

If you want to have sublists under a list item, you can do this by indenting the lines for the sublists. So you can write a list with a sublist like this:

```
* This is a top-level list item
    * Here is a sublist item
    * Here is another
* Now we are at the top level again.
```

Just add sufficient spaces to put the sub-items under the enclosing item. The result will look like this:

- This is a top-level list item
  - Here is a sublist item
  - Here is another
- Now we are at the top level again.

When you indent, you need at least four spaces or a tab per level.

Block Quotes

Should you need to add a quote to your text, you put a “>” before the quoted text.

So you can write:

```
> This is a blockquote. The blockquote
> can span multiple lines. If you don't
> put any new lines in it, you only
> need to put the ">" at the beginning
> of the line.
> If you want multiple lines where you
> include new lines, you should add
> the ">" to each line.
```

The result will look like this:

> This is a blockquote. The blockquote can span multiple lines. If you don't put any new lines in it, you only need to put the ">" at the beginning of the line. If you want multiple lines where you include new lines, you should add the ">" to each line.

Verbatim Text

Sometimes, you don't want any formatting at all of a text; you want to leave it verbatim as it is. When you want this, you can indent it with a tab (or four spaces). You can write this:

```
This will be shown
absolutely verbatim
```

The result will then look like this:

```
This will be shown
absolutely verbatim
```

Sometimes, you also want to add verbatim text inline in a paragraph. To achieve this, simply put the verbatim text in backticks. So you can write `'this'` to achieve `this`.

Links

Markdown was initially written to make it easier to write content for web pages. Consequently, it has built-in syntax for inserting hypertext links. These will work both with links to web pages and for cross-references within your text.

You have two options for specifying a link: you can put the destination URL where you insert the link, or you can create a label and map it to the URL so you can refer to the label when you insert a link. To put the URL where you insert the link, you put the text you want to be the link in square brackets and the destination URL for the link in round parentheses right after. You would write a text like this:

```
This is a link to [my blog](http://www.mailund.dk).
```

This is fine for most cases, but if you have many links in a paragraph, then the link annotations start interfering with how easy it is to read the text. Instead, you can give the destinations a shorter name and put the destination later in the text. To do this, you replace the round parentheses with square brackets, like this:

```
This is a link to [my blog][blog].
```

Then, later in the text, you define what the link should point to like this:

```
[blog]: http://www.mailund.dk
```

You use the same syntax to make hypertext links within your document. The simplest way to create a link to a section is to leave out the destination but put the name of the section in square brackets. A link to this chapter would then be written like this:

```
This is a link to the [Writing Markdown] chapter.
```

This, of course, will not work if you want the link to contain a different text than the section name, or if you have several sections with the same name. You can work around this by giving the section headers explicit labels. These, you put in curly brackets after the section header. You can assign a label to a header like this:

```
# My header {#header}
```

The hashtag is needed here and is also necessary when linking to the section. The hashtags are used in HTML to refer to sections of a web page, and it is from there that Markdown gets its syntax. To link to the section we have labeled this way, we would write the link like this:

```
This is a link to [the section](#header).
```

Links are only handy for hypertext documents, and in standard Markdown, you cannot make cross-references to figures or tables or any non-section elements. In Pandoc you can, using an extension, but we cover that in Chapter 7.

Images

To insert figures in your document, you use a syntax similar to inserting links. The difference is that you need to put a bang, "!", before the link.

```
![Title of the figure](URL-to-figure)
```

Typically, you will have the figures as local files and there you use the path to the figure file, either relative to where you build your document or as an absolute path.

```
![Title of the figure](path-to/my-figure)
```

Exercises

In the following text are a few exercises where you can test yourself on the material covered previously. If you use an editor that can immediately show you the result of a Markdown text, then test your results there. Otherwise, put your answers aside until we have covered how to format Markdown in Chapter 5.

Sections

Write a document with three level-two sections and with two level-three sections inside each. Remember that the number of # determines the section level. Make them both numbered and unnumbered.

Emphasis

Write *this* **text** in *Markdown*.

Lists

Take the three items

- one item
- two items
- three items

and write them as a numbered and an unnumbered list.

Now, put

- four items
- five items

as a sublist under *one item*.

Block Quotes

Make the following text a block quote:

```
This is a text that we want to put in block quotes.
Your task is to do this.
```

Links

Write a text and insert a link to a web page and to a section in your text.

Images

Find an image file and insert it into a text.

CHAPTER 4

Pandoc Markdown Extensions

Markdown is, unfortunately, not standardized. Different tools will support different markup syntax and process them differently. The Markdown described in the previous chapter will work in most, if not all, tools. The table syntax is usually less well supported, but the rest of the markup will ordinarily work.

Pandoc provides several extensions to the Markdown language described in the previous chapter. In this chapter, we will see some useful extensions for lists and tables. To get a complete list of Pandoc extensions to Markdown, you should consult the Pandoc documentation at `https://tinyurl.com/y87mstzf`.

Lists

Generally, the numbers you use when you write an ordered list are ignored. They are used to indicate list items, but the actual numbering does not matter. This makes it easier to insert a new item in the middle of a list, which is a good thing, but sometimes you want to start a list at a different number than one, and in that case, basic Markdown can't help

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_4

you. With Pandoc, though, lists start with the number you give the first item in a list. So you can start a list at number three like this:

```
3. This lists start at number three.
5. Although we used "5." to start this
   item, it still gets the number 4.
```

The numbers in the following list items are still ignored. The result will look like this:

> 3. This lists start at number three.
>
> 4. Although we used “5.” to start this item, it still gets the number 4.

This is a good way to continue lists, but you will have to update the initial number when you have added or removed items in the previous list.

To automatically make a list continue at the next number, even when you changed a previous list, you can use the special symbol “@”. This works just as a number when you use it in a list, and it always counts from where you left off. So you can write something like this:

```
(@) Starting a list
(@) Continuing the list

Here is some text that doesn't
belong to the list.

(@) This continues the list,
    numbered from where we
    left off the list.

(@label) This item is labelled
    so that we can refer back
    to it. Like this: see item
    (@label).
```

The result will look like this:

(1) Starting a list

(2) Continuing the list

Here is some text that doesn't belong to the list.

(3) This continues the list, numbered from where we left off the list.

(4) This item is labelled so that we can refer back to it. Like this: see item (4).

This can be very useful for lists of examples or such, but the "@" counter is global so you cannot restart the counter. Unfortunately, there is currently no support for both automatically numbered items *and* restarting counters.

In this example, we also saw another feature of Pandoc. We do not have to write a number followed by a dot. In the previous example, we put numbers—represented by @—in parentheses, and the result was a list that used parentheses. You can also use a single closing parenthesis:

```
1) This is a
2) list
3) it really is
```

It will give you this list:

1) This is a

2) list

3) it really is

If you make lists using a number and a period, you get the standard numbered lists, but you can also use letters or Roman numerals just by

starting the list with such. If you want to use parenthesis instead of periods, you can also do this. You can, for example, create a list like this:

a) This list uses letters instead of numbers.

b) We can make a sublist with roman numerals:

   i) This sublist also uses parenthesis

   ii) Cool, isn't it?

The source markup for that list looks like this:

a) This list uses letters instead of numbers.

b) We can make a sublist with a roman numerals:

   i) This sublist also uses parenthesis

   ii) Cool, isn't it?

You can mix the different list notations so you have different list formats as sublists, but if you mix them at the same level, you will start a new list.

Lists are frequently used to define terms or concepts, and in Pandoc you can create definition lists by following a term with a colon and indenting the start of the definition with a tab or at least four spaces. So you can create definition lists like this:

Something we want to define. Definition of the thing

As long as we indent the following lines, they become part of the definition.

Here starts the next thing we define. Here we write the definition.

More of the definition

The syntax for creating this list is this:

```
Something we want to define.
:    Definition of the thing

        As long as we indent the following lines, they
        become part of the definition.
```

```
Here starts the next thing we define.
:    Here we write the definition.

    More of the definition
```

An alternative syntax uses tildes, "~", instead of colons:

```
Term 1
  ~ Definition 1

Term 2
  ~ Definition 2a
  ~ Definition 2b
```

Term 1 Definition 1

Term 2 Definition 2a

Definition 2b

Either syntax will do.

Tables

To add tables to a document, you can mark up the columns using dashes. You can write a table like this:

```
  Right    Left     Center    Default
-------    ------ ----------  -------
     12    12        12           12
    123    123       123         123
      1    1          1            1
```

The result will look like this:

Right	Left	Center	Default
12	12	12	12
123	123	123	123
1	1	1	1

How you align the elements in the columns determine how you align the headers above the dashes. If the text is to the right, the column will be right-aligned. The same for left and center alignment, if the text is on the left or in the middle, the table column will be left-aligned or centered. If you start the header at the same position as the dashes, you get the default alignment, which is left alignment.

You can leave out the headers, but then you need to repeat the dashes at the end of the table as well.

```
-------     ------ ---------- -------
     12     12         12            12
    123     123        123          123
      1     1           1             1
-------     ------ ---------- -------
```

12	12	12	12
123	123	123	123
1	1	1	1

If you do not provide a header, the alignment is determined by the first line in the table. In the preceding example, in the last column, the reason it isn't right-aligned is that there is a space after the first number in the column before the end of the dashes that specify the column. Move the first number one position to the right, and that column would also be right-aligned.

Tables are the Markdown markups with the least consistent support in different tools. The table syntax described often frequently works, but in the two editors I use for writing my books, they are not supported. In all the Markdown viewers I am familiar with, though, they display correctly.

In Pandoc, there is excellent support for tables, and Pandoc provides some extensions to table markup beyond the preceding notation.

For figures, you can add captions using the link syntax. For tables, you have no similar syntax. You can, however, add a caption to a table using "Table: Caption" following the table. So you can create a table with a caption like this:

```
-------     ------ ---------- -------
     12     12        12           12
    123     123       123         123
      1     1          1            1
-------     ------ ---------- -------

Table: This is a caption
```

The result looks like this:

Table 4-3. *This is a caption*

12	12	12	12
123	123	123	123
1	1	1	1

You can leave out the "Table" part if you want; just having the colon will give you a caption.

If you want table cells to span multiple lines, you can do this as well. For multi-line tables, you must put a row of dashes before the header (unless you don't have a header), you must end the table with a row of

dashes and then a blank line, and you must separate rows by a blank line. This example, from the Pandoc manual, shows how this works:

```
-------------------------------------------------------
  Centred   Default      Right Left
   Header   Aligned    Aligned Aligned
----------- ------- ---------- ----------------------
    First   row            12.0 Example of a row that
                                spans multiple lines.

   Second   row             5.0 Here's another one.
                                Note the blank line
                                between rows.
----------- ------- ---------- ----------------------
```

The result will look like this:

Centred Header	Default Aligned	Right Aligned	Left Aligned
First	row	12.0	Example of a row that spans multiple lines.
Second	row	5.0	Here's another one. Note the blank line between rows.

You can leave out the header, but then you must repeat the dashes that define the columns after the table, followed by a blank line.

```
----------- ------- ---------- ----------------------
   First    row            12.0 Example of a row that
                                spans multiple lines.
  Second    row             5.0 Here's another one.
                                Note the blank line
                                between rows.
----------- ------- ---------- ----------------------
```

Result:

First	row	12.0	Example of a row that spans multiple lines.
Second	row	5.0	Here's another one. Note the blank line between rows.

An alternative syntax for tables uses plain text grids to separate columns. An example, with a header, looks like this:

```
+---------------+---------------+--------------------+
| Right         | Left          | Centered           |
+==============:+:==============+:==================:+
| Right         | Left          | Centered           |
+---------------+---------------+--------------------+
```

Result:

Right	Left	Centered
Right	Left	Centered

Without headers, it looks like this:

```
+--------------:+:--------------+:-----------------:+
| Right         | Left          | Centered          |
+---------------+---------------+-------------------+
```

Result:

Right	Left	Centered

For these types of headers, the colons determine the alignment. Put a colon at the end of the "=" or "-" of the cell-border to get right-aligned columns, at the left to get left-aligned columns, and at both ends to get centered columns, and leave them out for the default alignment.

You can also use pipes, "|", to specify the columns. Then, the syntax will look like this:

```
| Right | Left | Default | Center |
|------:|:-----|---------|:------:|
|   12  |  12  |    12   |    12  |
|  123  |  123 |   123   |   123  |
|    1  |    1 |     1   |     1  |
```

Result:

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

Smart Punctuation

Proper text typography uses different types of dashes—hyphens in words, en-dashes for numeric intervals, and em-dashes for parenthetical sentences and emphasis. Quotes are usually different at the beginning and end of a text in quotation marks. Three dots are different from ellipses. Many word editors will automatically substitute some dashes and translate straight quotes into the correct form, but not all. Since you are writing Markdown in plain text, and since most keyboards do not give you access to all typographic symbols, Pandoc can help you out with getting these symbols. To enable this, you need to call Pandoc with the option `--smart` (we see how to invoke Pandoc in Chapter 5).

If you turn on smart punctuation, quotes will be handled correctly, three dots will be translated into ellipses, a single "-" will be a hyphen, two dashes will give you an en-dash, and three dashes will give you an em-dash.

Footnotes

You can insert footnotes in your text using a syntax resembling links and images. For footnotes, you need to add a caret, (^). To put the footnote inside the text paragraph you are writing, you add a caret and then the footnote text in square brackets.

```
Footnote inside a paragraph.^[This is the footnote.]
```

Since footnotes tend to interfere with the main text, you can give them a label and add the text elsewhere. When you do this, you name the footnote and put the name in the main text.

```
Reference to a footnote.[^footnotelabel]
```

When you add a footnote this way, the caret has to go inside the brackets. If it goes before the brackets, you are adding the footnote inside the text.

Somewhere the text you must define what a footnote label refers to. The syntax is the same as the one for defining links you can refer to inside the text, except that for defining footnotes the name must start with a caret.

```
[^footnotelabel]: This is footnote two.

    If you want it to cover multiple lines, you
    have to indent the following footnote lines.
    by at least four spaces.
```

The output of the two approaches should look like this: Footnote inside a paragraph.[1]

Reference to a footnote.[2]

[1]This is footnote one.

[2]This is footnote two.
If you want it to cover multiple lines, you have to indent the following footnote lines by at least four spaces.

Exercises

Lists

Write a list with five items. Between two and three, add a paragraph of text, but make sure that the numbers continue with four.

Add a label to item three and refer to it in item five. Create a list of definitions as well. Make them multi-line.

Tables

Make a table with three columns where you left-justify the first column, center the second column, and right-justify the third.

Write a table with a caption. Until we format the Markdown with Pandoc in the next chapter, you will not see the result but keep the text so you can test it there

Footnotes

Write a text with a footnote. Use both notations for footnotes.

CHAPTER 5

Translating Documents

Once we have a document written in Markdown, we want to translate it into other file formats using Pandoc. First, though, we have to download Pandoc. Go to Pandoc's installation guide at http://pandoc.org/installing.html and follow the instructions relevant for your platform.

Formatting a Markdown Document with Pandoc

For our first example, we can take the small Markdown document shown here:

```
# This is a test document

Here is some text in the document.

* This is a list
* With two items
```

If we save this Markdown document in a file called `input.md`, we can translate it into an HTML file, `output.html`, using the command:

```
pandoc -o output.html input.md
```


T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_5

The `-o` option specifies the output file. The `input.md` file is specified without any options. You do not need any options for input files, and you can provide more than one. If you provide more than one input file, they are in effect concatenated before Pandoc processes them, so if you want to construct a book from several chapters you have written in separate files, you can provide them on the command line in the order you want the chapters to appear in the book.

Pandoc figures out the input and output format from the file extensions, so if you use the preceding command, it will know that the input is Markdown (filename suffix `.md`) and that the output should be HTML (filename suffix `.html`). You can make the format of input and output formats explicit. You can use the option `--from` to specify the input format and `--to` to specify the output format. In most cases, you will not need to specify the formats—the filenames contain all the information you need—but sometimes different formats share the same filename suffixes, such as the EPUB and EPUB3 formats that both use filename suffix `.epub`. In those cases, you need the options.

If you specify the input and output document format, then you can also treat `pandoc` as a program you can pipe input into and get the formatted document out from. You could, for example, write

```
cat input.md | \
    pandoc --from markdown --to html \
    > output.html
```

In itself there is little use for this, but combined with preprocessing (Chapter 10) and filters (Chapter 11), it is very handy.

Back to the output of `pandoc`. If you run the previous command

```
pandoc -o output.html input.md
```

in your terminal, then the `output.html` file should now contain the following HTML:

```
<h1 id="this-is-a-test-document.">
    This is a test document
</h1>
<p>Here is some text in the document.</p>
<ul>
<li>This is a list</li>
<li>With two items</li>
</ul>
```

If you are not familiar with HTML, this might not be readable, but I hope that you can at least recognize the elements from the input Markdown.

This HTML is not a complete HTML file. It is a fragment of an HTML file that corresponds to the Markdown document, but it is missing header and footer markup that is needed for a complete HTML page. Per default, Pandoc creates HTML markup that can be added to a web page, but not standalone documents. To get the header and footer added as well, you can use the option `--standalone`.

```
pandoc --standalone -o output.html input.md
```

The `--standalone` option is needed for HTML output if you want a complete document. If you choose an output format that is typically not meaningful as a fragment, such as PDF documents (suffix `.pdf`), EPUB documents (suffix `.epub` or `.epub3`, or Word files (suffix `.docx`), Pandoc will automatically create complete documents, and the `--standalone` option is not needed.

If you run `pandoc --standalone -o output.html input.md`, you will get a warning:

```
[WARNING] This document format requires a nonempty
  <title> element.
```

```
  Please specify either 'title' or 'pagetitle' in
  the metadata,
  e.g. by using --metadata pagetitle="..." on the
  command line.
  Falling back to 'input'
```

But despite the warning, you will get an HTML document that contains all the elements such a document needs:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      lang="" xml:lang="">
<head>
  <meta charset="utf-8" />
  <meta name="generator" content="pandoc" />
  <meta name="viewport"
        content="width=device-width,
                 initial-scale=1.0,
                 user-scalable=yes" />
  <title>input</title>
  <style>
      code{white-space: pre-wrap;}
      span.smallcaps{font-variant: small-caps;}
      span.underline{text-decoration: underline;}
      div.column{display: inline-block;
                 vertical-align: top; width: 50%;}
  </style>
</head>
<body>
<h1 id="this-is-a-test-document">
    This is a test document
</h1>
```

```
<p>Here is some text in the document.</p>
<ul>
<li>This is a list</li>
<li>With two items</li>
</ul>
</body>
</html>
```

There is more general HTML here than there is in our text, but luckily we do not need to worry about it; we can focus on the Markdown and let Pandoc worry about the rest.

The warning we got has to do with the title we get in the line

```
<title>input</title>
```

Pandoc just took the name from the input file but was hoping to get the title explicitly provided. You can do this using metadata; see Chapter 8. Ignore it for now.

You can try making different output formats with the commands

```
pandoc -o output.pdf input.md
pandoc -o output.docx input.md
pandoc -o output.epub input.md
pandoc --to=epub3 -o output.epub input.md
```

In the last example, we explicitly specify that the EPUB format to use in the output is EPUB3.

Another case is where we might want to specify the --to format is for PDFs. By default, Pandoc will create PDF files using LaTeX, but you can specify that it should use ConTeXt instead with the command:

```
pandoc --to=context -o output.pdf input.md
```

To get a complete list of supported input and output formats, run the commands

```
pandoc --list-input-formats
```

and

```
pandoc --list-output-formats
```

respectively.

Pandoc can translate between many different input and output formats, but in this book, we will only consider Markdown input and how to translate Markdown to other formats.

Frequently Useful Options

There are many options you can use to influence how Pandoc transform a document. I refer you to the online manual[1] for a full list. Here, I will list a few that I find particularly useful in my own writing.

Sections and Chapters

First, we consider options that relate to how sections are interpreted. In Markdown we specify the different levels of section headers by the number of hashtags, but when we produce a document, we sometimes have to worry about whether the top-level sections are parts, chapters, or sections. If we are writing a short report or paper, we want the level-one headers to start sections, but if we are writing a book, we want them to start chapters. The default depends on the output we produce, and Pandoc does some guessing for us, but you can choose explicitly what

[1] http://pandoc.org/MANUAL.html

the top level should be using the `--top-level-division` option. For my books, where I want the level-one headers to be chapter headers, I use `--top-level-division=chapter`.

Table of Contents

To add a table of contents to your output document, you can use the option `--toc` or the option `--table-of-contents`; the first is just a shorter version of the second. You can specify the level of sections you want in the table of contents using the `--toc-depth` option. When I produce ebooks, I typically only want the table of contents for the chapter level, so I use `--toc-depth=1`. When I produce PDF, I am happy with the default. You can play around with the option to see what you prefer.

Image Extensions

If you want to produce both PDFs and ebooks from your Markdown input, you might want to use PDF vector graphics for figures for the PDF output but bitmap PNG for the ebook version. Using bitmap graphics for the PDF output means you have to worry about the resolution, but using PDF graphics for ebooks doesn't always work. When you insert images into your document using the

```
![Figure caption](graphics-file)
```

syntax, you need to specify the input file name, but if you want to produce both ebooks and PDFs, you don't want to have to change all the file names depending on which output format you are producing.

You can leave out the filename suffix for graphics files and specify the desired suffix using the

```
--default-image-extension
```

option instead. Then, for any graphics file where you haven't explicitly written the filename suffix, Pandoc will use the default. I always use

```
--default-image-extension=pdf
```

when producing PDF documents and

```
--default-image-extension=png
```

when producing ebooks.

Ebook Covers

Ebooks contain cover images together with their text. If you produce ebooks, you want to specify the cover image as well. You can do this using the `--epub-cover-image` option. If your cover image is in the file `cover.png`, you write

```
--epub-cover-image=cover.png
```

Using Makefiles

For this book, I have all my text in a single `book.md` document. This is fine for a short book like this one, but usually, I keep each chapter in a separate file. The command line for compiling my books can get rather long, and I often have various options for different output formats that I need to remember, so if I had to use the command line each time I wanted to build a new version of a book, it would quickly become tedious and extremely error-prone. So I use Make (https://tinyurl.com/nyc2ec2) for compiling my books.

If you are not familiar with Make, I will give you sufficient detail for to read the Makefile I give as an example, and maybe a starting point for your own Pandoc Makefile, but introducing Make in full detail is beyond

the scope of this book. Many programs solve the same problem that Make does, so there are alternatives to choose from if you do not like Make.

The Makefile I use is not sophisticated and it suffices to know this:

1. You define a variable by writing

   ```
   VARIABLE_NAME := VALUES
   ```

2. You refer to the value that a variable holds using

   ```
   $(VARIABLE_NAME)
   ```

3. When you write `target: dependencies` you say that of you want to have `target` and then, if any of the dependencies have changed since lasts time you constructed `target` then you need to construct it again. If dependencies of dependencies have changed, then you need to build the dependencies and then the target. And so on.

4. The lines after `target: dependencies` that are indented by a tab are the instructions your computer needs to make `target`.

5. The first `target: dependencies` line in the Makefile is the target that Make will handle if you do not provide another target on the command line.

The Makefile I use for this book looks roughly like this (although I have left out a few options). I will walk you through it here.

```
CHAPTERS := header.yml book.md
PANDOC := pandoc

OPTS_ALL := --toc --smart \
            --top-level-division=chapter
```

```
PDF_OPTS := $(OPTS_ALL) \
            --default-image-extension=pdf

EPUB_OPTS := $(OPTS_ALL) \
            --default-image-extension=png \
            -t epub3 --toc-depth=1 \
            --epub-cover-image=cover.png

all: book.pdf book.epub book.docx

book.pdf: $(CHAPTERS) Makefile
    $(PANDOC) $(PDF_OPTS) -o $@ $(CHAPTERS)

book.epub: $(CHAPTERS) Makefile
    $(PANDOC) $(EPUB_OPTS) -o $@ $(CHAPTERS)

book.docx: $(CHAPTERS) Makefile
    $(PANDOC) $(PDF_OPTS) -o $@ $(CHAPTERS)

clean:
    rm book.pdf book.epub book.docx
```

I use a variable to hold the input files. Just the header and the single Markdown file in this case. You can specify metadata (see Chapter 8) on the Pandoc command line or in a YML file. You can put the metadata in your Markdown text, but then it has to be at the top of the input, and you cannot translate a single chapter without including the first one. I prefer to put my metadata in a separate file, which in this case is `header.yml`.

I only have one Markdown file for this book. I am writing this in the Ulysses editor where I can split the book into different sections but export it as one combined file. Since the partition into chapters and sections is kept in Ulysses and not in separate files, I do not have a file per chapter.

I put the input files in the variable `CHAPTERS`. If you have several, you can add it to the `CHAPTERS` variable. I keep the Pandoc command tool in a variable as well. I have more than one version installed, and I can switch

between them by updating the variable. After that, I put the arguments that all Pandoc runs share in `OPTS_ALL` and then PDF and EPUB-specific options in `PDF_OPTS` and `EPUB_OPTS`. I make a Word document, but I can use the PDF arguments for this; Pandoc will know how to make a Word document with the same options as used for the PDF. Next is all the targets, dependencies, and commands for making the targets. The targets `all` and `clean` are special. The first doesn't build anything; it just triggers a build of its dependencies—so it is indirectly building these—which are the three book formats the Makefile knows how to make. The `clean` target doesn't have any dependencies, but it will delete the books we generate with the middle three commands.

I then define some options I want to use for all output formats and then options that I only want to use for PDF and others I only want to use for ebooks. In the metadata I set for PDF output, I could also have put in the header—they wouldn't interfere with the EPUB output if I did—but I have chosen to set them here.

I build an EPUB book in the Makefile, but if you are planning to publish on iBooks, I do not recommend using this file. I find it much easier to format and submit a book using Pages. Pages can read the Word document, and that is how I usually submit a book to iBooks: I make a Word document, open it in Pages, and then submit it. You might think that building a book in the right format and then use iTunes Connect to submit it would be easier. You would be wrong. If you want to publish on Amazon (on Kindle Direct Publishing), you are also better of using their tool Kindle Create. Kindle Create can read the Word file, and you can submit from there. There is a command line tool that can translate from EPUB to the MOBI file format used on Kindle, but Kindle Create is easier to use.

CHAPTER 6

Math and Computer Programming Languages

You might not be in the habit of writing technical or scientific text, but if you are, then Pandoc supports both mathematics and computer programming code formatting.

Writing Math

Pandoc has some excellent support for writing math as long as your output file format supports it. Word files have built-in support for math, and Pandoc will use this if the output file is a Word document. With PDF, Pandoc uses LaTeX as an intermediate format, and LaTeX is spectacularly perfect for math. With HTML and EPUB, you have some math support depending on the version and libraries you use to display the math.

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_6

For HTML and EPUB, you have different options for how the output should handle math. I usually use either MathML[1] or MathJax[2] when I have math in HTML or EPUB documents. For MathML you can use the option `--mathml`:

```
pandoc --mathml -o document.epub document.md
```

For MathJax, you need to produce EPUB3 documents to get it to work, so you must specify the output format as well:

```
pandoc --to=epub3 --mathjax \
       -o document.epub document.md
```

With either option, you get an EPUB document that displays the math well. Not as beautiful as in PDF documents, where LaTeX handles the math, but reasonably well.

The way you write math in your document is by inlining TeX syntax. The math in output documents will only be TeX if you output a LaTeX file or go through LaTeX when building a PDF file, but the de facto standard for how to write math in plain text is TeX, so that is what Pandoc uses.

You start and end math using dollar signs. If you use one, you get inline math, and if you use two, you get "display" math, which means you get the math on a single line and centered. This is inline math: `$\int_a^b x\, \mathrm{d}x$` looks like $\int_a^b x\, \mathrm{d}x$. You write display mode math between double dollar signs, so this

```
$$\sum_{j=1}^N \frac{1}{j^2}$$
```

will result in this:

$$\sum_{j=1}^{N} \frac{1}{j^2}$$

[1] www.w3.org/Math

[2] www.mathjax.org

Pandoc supports a large subset of math you can write in LaTeX. For example, here is a more complex equation:

$$d(i,j) = \begin{cases} i & j = 0 \\ j & i = 0 \\ \min\begin{cases} d(i-1,j-1) + 1(i \neq j) \\ d(i,j-1) + 1 \\ d(i-1,j) + 1 \end{cases} & i > 0,\ j > 0 \end{cases}$$

The LaTeX code looks like this:

```
$$
d(i,j) = \begin{cases}
\,i & j = 0 \\
\,j & i = 0 \\
\,\min\begin{cases}
    d(i-1,j-1) + \mathrm{1}(i\neq j) \\
    d(i,j-1) + 1 \\
    d(i-1,j) + 1
\end{cases} & i>0,j>0
\end{cases}
$$
```

If you write math that Pandoc cannot handle, for example, if you use a LaTeX package that Pandoc does not know about, then you can just write the LaTeX code anyway. What Pandoc doesn't understand it passes on to the output. So, as long as your output is a format that goes through LaTeX, you are good to go.

If your output is HTML or EPUB where you use MathML or MathJax, then you can also write an equation that should be passed along verbatim to the output.

If you have a single document but want separate equations for different output formats, you cannot just let the text fall through to the output. There, you need to output one version for one output format and another for another format. Here, Pandoc lets you write verbatim text that will only be included for a specific output format. You can write

```
```{=latex}
$$\sum_{j=1}^N \frac{1}{j^2}$$
```
```

for a block of text that should only be included if the output is LaTeX and write

```
`$f(x)=x^2$`{=latex}
```

for inline text.

Writing Code Blocks

Markdown is frequently used by programmers to write about programs, so not surprisingly it has support for displaying code examples, typically with formatting to syntax highlight the examples. This support varies between different tools, but Pandoc has terrific support for many programming languages.

The syntax for writing a block of code uses either tilde, "~", or backticks, "`". Of these, backticks are more likely to be supported by other tools; so, I will use these here. Every example I give in the following will also work if you use ~ instead of a backtick.

You start a block with three backticks, then write your code, and then end the block with another three backticks.

```
```
for (int i = 0; i < n; i++) {
    printf("%d\n", i);
}
```
```

You can also use tiles instead of backticks, but I will use backticks in this chapter.

```
~~~
for (int i = 0; i < n; i++) {
    printf("%d\n", i);
}
~~~
```

If you only use backticks or ~, you get a verbatim text block, just as if you had indented the text. To get syntax highlighting, you must also inform Pandoc of which programming language you are using. You can do this by writing the name of the language after the first line of backticks. To syntax highlight the preceding code, which is written in C programming language, you would write

```
```c
for (int i = 0; i < n; i++) {
    printf("%d\n", i);
}
```
```

The result will look like this:

```
for (int i = 0; i < n; i++) {
    printf("%d\n", i);
}
```

A block of Python code would look like this:

````
```python
for i in range(n):
    print(i)
```
````

The result would look like this:

```
for i in range(n):
    print(i)
```

To get a complete list of supported programming languages, you can run the command

```
pandoc --list-highlight-languages
```

Code Block Options

This syntax for displaying code is supported by many web sites where it is usual to write code in text, such as on GitHub. You can give more advanced instructions for how code should be displayed, but then you need to use a slightly different syntax that is more Pandoc specific. There, you put instructions in curly brackets after the first three backticks. You must specify the programming language but prepend a dot to the name, for example, write `.python` for Python. You can then number the code lines with the instruction `.numberLines`:

````
```{.python .numberLines}
for i in range(n):
    print(i)
```
````

The result will look like this:

```
1   for i in range(n):
2       print(i)
```

You can specify which line number to start from with the option `startFrom`. To start from line 100, we can write

````
```{.python .numberLines startFrom=100}
for i in range(n):
    print(i)
```
````

The result will look like this:

```
100  for i in range(n):
101      print(i)
```

You can give the code block an identifier:

````
```{#my-code .python .numberLines}
for i in range(n):
    print(i)
```
````

This will make the code block a hypertext target (an anchor in HTML), so you can create hypertext references to it. The anchor id is the same as the identifier excluding the hashtag. That is, in the preceding example, it is `my-code`. You refer to the target as you would with other references in a Markdown document, for example:

````
```{#my-code .python}
for i in range(n):
    print(i)
```

See [my code](my-code)
````

In HTML output, you can also make each line an anchor using the option `.lineAnchors`.

````
```{#my-code .python .numberLines .lineAnchors}
for i in range(n):
    print(i)
```
````

The line identifiers are the code block id followed by the line number. In this case, the two lines `my-code-1` and `my-code-2`.

In HTML output, all the options that start with a `.` will be added to the class of the `<pre>` block that contains the code. With a stylesheet, you can easily make code blocks formatting distinct. In filters (see Chapter 11), you can use the options for arbitrary transformations or calculations. One crucial thing to keep in mind if you start using code block options to a larger extent is that all options without arguments

```
.python
.numberLines
```

must start with a dot, and all options that take an argument

```
startsFrom=100
```

cannot start with a dot. If you break these rules, Pandoc will not do what you want.

Syntax Highlighting Styles

The syntax highlighting scheme is controlled by variations of stylesheets: cascading style sheets (CSS) for HTML output and a set of `\newcommand` options for LaTeX (and thus PDF) output. These highlighting instructions are put directly in the output file (when you make a standalone document) and are thus not easy to override. You can, however, choose

from a list of predefined highlighting styles. You can see the complete list by running the command

```
pandoc --list-highlight-styles
```

You select a style using the `--highlight-style` option. When I build print versions of my books, I want a black-and-white output, so I use the option

```
pandoc --highlight-style=monochrome
```

This gives me a black and white highlighting using italic and boldface to display different language components. You can also completely disable syntax highlighting while still using code blocks, using the option `--no-highlight`.

Exercises

Code blocks

Write the Markdown to display this Python code:

```
print("hello, world")
for i in range(10):
    print(i)
```

Code Block Options

Now add line numbers to it

Syntax Highlighting

Format the code in at least three different highlight formats and check the results.

CHAPTER 7

Cross-referencing

Cross-referencing is not directly supported by standard Markdown. Markdown was mainly invented to write hypertext documents for web pages, so referencing section numbers, figures, or tables is not part of it. We can use the link syntax to make hypertext references to sections but not get section numbers and such.

Pandoc itself doesn't support extensions for other kinds of cross-referencing, but there is a so-called "filter," `pandoc-crossref`, that adds this support. Filters (see Chapter 11) are scripts that are run to modify a document after it has been parsed by Pandoc and before the output format is generated. Filters are beyond the scope of this book, but we will use two in this and the next chapter.

The `pandoc-crossref` filter is not necessarily automatically installed when you install Pandoc, so you might need to install it manually. How you do this will depend on your platform. I work on MacOS and use the Homebrew[1] package manager, so I installed it using

```
brew install pandoc-crossref
```

The filter is a Haskell package, so you can also install it using the `cabal` package manager if you have the Haskell system installed.

```
cabal update
cabal install pandoc-crossref
```

[1] http://brew.sh

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_7

It might take a little while.

If you installed Pandoc using any of the packages from the Pandoc releases web page,[2] the filter should already be included. If all else fails, you can download the source code and compile it at the pandoc-crossref homepage.[3]

To invoke the filter when processing a document with Pandoc, you use the option `--filter` (or `-F` as a shorter option). To enable the `pandoc-crossref` filter, run Pandoc as

```
pandoc --filter pandoc-crossref
```

Cross-referencing using `pandoc-crossref` uses a syntax similar to hypertext links, but with a twist. You specify labels with curly brackets and insert references to them using square brackets. You need to include cross-referencing type as part of the labels, though. To insert a label, you use the syntax `{#type:label}`, and to refer to it, you use the syntax `[@type:label]`. The type, here, specifies whether you are referring to sections, tables, figures, or equations.

Referencing Sections

Using cross-referencing via links works well for HTML or EPUB documents, where you have hypertext, but less well for printed media, where you don't. There, you would make references to chapters and sections using their numbers instead, but that isn't supported in standard Markdown.

Using `pandoc-crossref`, we can refer to sections via section numbers. We need to define labels for the headers we want to refer to, and these labels must start with the prefix `sec:` (in addition to the hashtag that all

[2] https://github.com/jgm/pandoc/releases
[3] https://hackage.haskell.org/package/pandoc-crossref

header labels must have). We can then insert cross-references to section numbers using `[@sec:label]` markup. An example could look like this:

```
# This is a chapter {#sec:chapter}

## This is a section {#sec:section}

See [@sec:chapter] and [@sec:section].
```

Pandoc doesn't number sections by default, and while it will number sections in some formats when we use `pandoc-crossref`, we should use the option `--number-sections` to make sure. If, for example, you generate PDF output, sections will not be numbered if you leave out this option, and consequently, it makes no sense to refer to sections by their numbers.

```
pandoc --number-sections \
    --filter pandoc-crossref \
    -o output.pdf \
    input.md
```

This approach will use the LaTeX section numbering system for PDF output but will also work for other output formats such as HTML.

You can set the section numbering depth with the metavariable `secnumdepth`. For example, to number chapters and sections (or sections and subsections, depending on the top level section type), you can add the following line to your metadata header:

```
secnumdepth: 2
```

This will only affect LaTeX and PDF output, though, and not other output formats.

Alternatively, you can leave the section numbering entirely up to the filter by setting metadata variable

```
numberSections
```

to `true` and set the section depth you want to be numbered with the metadata variable

`sectionsDepth`

For example, in your header, you can add the following lines to get chapters and sections, but not subsections, numbered (assuming the top level headers are chapters; otherwise you get sections and subsection headers numbered):

```
numberSections: true
sectionsDepth: 2
```

This will insert chapter and section numbers at the desired depth and let you cross-reference sections in HTML and EPUB format, but unfortunately the cross-referencing does not work in LaTeX and PDF. Here, you only get the section numbers but not the cross-references inserted.

Neither of the two choices for section numbering is ideal for both PDF and EPUB output. Using

`--number-sections`

will insert section numbers in both output formats, but you can only control the numbering depth in PDF output. Using

`numberSections`

will insert section numbers to the desired depth in both output formats, but you can't insert references to them in PDF output. You can't use *both*

`--number-sections`

and

`numberSections`

either since both the Pandoc filter *and* LaTeX will insert section numbers, and you end up with two of them in each header.

One solution I use for this is to call Pandoc with different options when generating EPUB and PDF. For EPUB output, I use

```
pandoc --metadata numberSections=true \
       --filter pandoc-crossref ...
```

For PDF output, I use

```
pandoc --number-sections \
       --filter pandoc-crossref ...
```

I set the numbering depth in my YAML header, using both options for controlling that:

```
sectionsDepth: 2
secnumdepth: 2
```

Reference Prefixes

When you need to refer to a section, you use the syntax

```
[@sec:label]
```

When you reference a section this way, the filter will insert both the section number and a default prefix, which is "sec." for a single section and "secs." for multiple sections. You can refer to more than one section but separating the labels by semicolons inside the square brackets:

```
[@sec:label1; @sec:label2]
```

For documents where you have both chapters and sections, this might not be what you want. There you might want to use the prefix "Chapter" for chapters and "Section" for sections. Unfortunately, you cannot make prefixes that depend on the header depth, but you can disable the prefixes by overriding them.

You can either change the reference prefix on a per-reference basis or globally through metadata. For referring to chapters as "Chapter" rather than "sec."—as in this example—the best solution is probably to set the prefix explicitly when referring to a chapter, but we can see how both approaches work.

To use a per-reference specific prefix, you need to insert the prefix you want between the start square bracket and the label. So, to make the prefix for the reference to a chapter be "Chapter," we would write `[Chapter @sec:chapter]`.

To change the prefixes globally, we need to set a metadata variable. The metadata variable that controls the section references prefix is `secPrefix`. If we set it to the empty string, we get rid of the prefixes.

```
secPrefix: ""
```

You can then manually insert the prefixes you want in the Markdown text:

```
See Chapter[@sec:chapter]
and Section[@sec:section].
```

Notice the lack of spaces between prefix and references here. This is needed for PDF output; the LaTeX document that Pandoc generates for the references contain a hard space, so if we put a space between the prefix and reference, the PDF document will have too much space in the generated text. For HMTL and EPUB, it doesn't matter.

Completely disabling a prefix can be done on a per-reference basis as well. Just add a "-" between the start bracket and the label. If you write `[-@sec:chapter]`, you only get the chapter number and not the prefix. You rarely need to set the default prefix to the empty string explicitly. But using that as an example gave me an excuse to introduce the variable.

You can set the `secPrefix` metadata to a list. The first element is used for single references and the second for plural. So, to use "Sect." as

the prefix for a reference to a single section, and "Sects." as the prefix for multiple sections, we could specify this metadata:

```
secPrefix: ["Sect.", "Sects."]
```

The prefix list can have any length, and the number of references is used to select a prefix. So, if you want a special prefix when you refer to three sections, you can add a third element to the list. When you have more references than prefixes, you get the last element in the list. Thus, if you specify two elements in the list, the first is used for singular references and the second for multiple references.

You might want to use lowercase "sect." when you refer to a section in the middle of a sentence but "Sect." at the beginning of sentences. With `pandoc-crossref` it is simple to switch between uppercase and lowercase label prefixes. If you insert a label that starts with an uppercase, your prefix will be in uppercase as well. Thus, if you write `[@Sec:label]`, the default prefix will be "Sec."; if you write `[@sec:label]`, the default prefix will be "sec.". The same goes for references to figures, tables, equations, and so on.

Referencing Figures, Tables, and Equations

To reference figures, you use the same syntax as for sections, except for where you define figure labels and the prefix of the labels. To define a label for a figure, you must give it the prefix `#fig:` and place the label right after the markup for inserting the figure, so using the syntax

```
[Caption](link-to-figure){#fig:label}
```

You cannot put a space between the figure insertion and the label definition. You refer to figure labels as you would with references to sections.

For tables, your labels must start with `#tbl:` and placed after the table caption:

```
 a   b   c
--- --- ---
 1   2   3
 4   5   6

: Caption {#tbl:label}
```

For tables you *must* have a space between the caption and the label.

For display-style math, math that stands on a line of its own and is written with two dollar signs, you can add labels if they begin with `#eq:` and are put on the same line as the math with a space between the label and the terminating double dollar signs:

```
$$ f(x) = x^2 + a x $$ {#eq:label}
```

You can change the default prefix for figures, tables, and equations, as you can for sections, with the metadata `figPrefix`, `tblPrefix`, and `eqnPrefix`, respectively.

For more details on how to use the cross-reference filter, I will refer to its online manual.[4]

Bibliographies

If your text requires citations and a bibliography, you can enable the filter `pandoc-citeproc` by either running Pandoc with the

```
--bibliography
```

option or using

```
--filter pandoc-citeproc
```

[4] https://hackage.haskell.org/package/pandoc-crossref

If you use both `pandoc-crossref` and `pandoc-citeproc` filters, you must always use `pandoc-crossref` first

```
pandoc --filter pandoc-crossref \
       --filter pandoc-citeproc ...
```

The two filters use similar kinds of citation syntax, and this means that the order in which you run the filters matter. The `citeproc` filter gets confused if it sees cross-reference labels. The same does not happen if you run the cross-reference filter first; it will leave the citation codes alone so they can be handled by the citation filter.

With the `--bibliography` option, you need to specify the file that contains your bibliography. Using

```
--filter pandoc-citeproc
```

you can specify the bibliography file as metadata in your YAML header instead, for example:

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

The `pandoc-citeproc` filter can read bibliographies in various file formats and will pick the format based on the file suffix. With `.bib` files it will use BibTeX. For EndNote you would use `.enl` and for ISI `.wos`. Check the online documentation[5] for a complete list.

To control the format used for citations and the bibliography, you can specify a CSL[6] file with the metadata variable `csl` or the Pandoc option `--csl`. CSL files for most journal styles can be downloaded from GitHub.[7]

For example, if "smith12" is a key in your bibliography, you can insert a citation using `[@smith12]`. You need to include `@` even though it isn't part

[5] https://tinyurl.com/yyx6j3aa
[6] http://citationstyles.org
[7] https://tinyurl.com/ac8f8eg

of the reference key; the filter (and Pandoc) uses this to recognize citations. You can cite more than one reference by separating the references with semicolons: `[@smith12; @smith14]`.

Depending on the citation style, the inserted reference might contain author names. This doesn't read well if you have already mentioned authors in the text, and you can disable it with a minus before the reference: `[-@smith12]`. The same effect is achieved by leaving out the square brackets and write `@smith12`.

More generally, you can insert text in the citations to add, for example, page information or other comments, such as `[see @smith12, chap1; also @smith14, chap 12]`. If you leave out the square brackets to get an in-text citation, you can add comments to appear inside the parenthesis (again, depending on your citation style) by writing the comments in square brackets after the reference: `see @smith12 [chapter 11]`.

The bibliography will be put at the end of your document. If you want to give the bibliography a section header, you should end your Markdown document with the header for the bibliography.

Exercises

Reference Sections

Write three sections. In two and three, refer back to one and two, respectively.

Figures, Tables, and Equations

Write a document with each of a figure, a table, and an equation. Make references to each.

Bibliographies

If you have a bibliography, then make a document that cites the elements in it.

CHAPTER 8

Metadata

If you go back and look at the standalone HTML document we looked at in Chapter 5, the document was this:

```
# This is a test document

Here is some text in the document.

  * This is a list
  * With two items
```

We compiled it like this

```
pandoc --standalone -o output.html input.md
```

You should get the warning

```
[WARNING] This document format requires a nonempty
<title> element.
Please specify either 'title' or 'pagetitle' in
the metadata,
e.g. by using --metadata pagetitle="..." on the
command line.
Falling back to 'input'
```

and the title in your document will then, just as the warning says, look like this:

```
<title>input</title>
```


T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_8

Pandoc inserted a title to your document, but it is set to a default value, input, because we didn't specify it. Try running this command instead:

```
pandoc --metadata title="My Title" \
       --standalone -o output.html input.md
```

If you now read the `output.html` file, you will see that Pandoc has inserted "My Title" between the title tags and inserted a level-one header that says "My Title".

When Pandoc generates a standalone document, it uses metadata such as title and author(s) to fill in some information. This data is usually not specified in the Markdown input—there aren't any Markdown annotations for defining such metadata—but you can set it using the `--metadata` option or using YAML (see the following text).

Strictly speaking, there are two types of variables that are used when producing the output: metadata, specified with `--metadata`, and variables, specified with `--variable`. The difference between them is that metadata can be seen and processed by Pandoc and Pandoc filters—scripts that process your input before it is formatted for the output—while variables are used in templates. If you set a variable using the `--metadata` tag, or in a metadata header, the variable will also be available to templates, so you can usually stick to metadata. The output isn't *exactly* the same since filters might do something with metadata that they won't do with variables, but it is easier to stick with one kind of options. So unless you have good reasons not to, use metadata.

YAML for metadata

There are potentially many values you want to specify as metadata, so you don't want to rely on command line options for all of those. Luckily, Pandoc can read metadata from a header in your input, specified in another markup language called YAML (Yet Another Markup Language).

YAML is a different kind of markup language than Markdown. It is not intended for marking up a text but for providing structured data to tools.

You can put a YAML header with metadata at the top of your input text to provide Pandoc with theinformation. I usually put my metadata in a separate file instead and give that as the first input file when I run Pandoc. Since Pandoc concatenates the input files you give it, this is equivalent to putting the metadata at the top of the document, but it does give me the option of using different headers when I produce output in different formats and I can easily format different Markdown files with the same metadata.

A YAML header starts with three hyphens `---` on a line of their own and is terminated with another three hyphens. Inside the header, you can put key-value information. The keys are followed by a colon, and the values follow the colon. The header I use for this book looks like this:

```
---
title:
  "My Markdown and Pandoc book"
author:
  - Thomas Mailund
year: 2019
---
```

It sets three values, the title, the author, and the year I am writing the book, which is all that I need for this book. I didn't need to put the title in quotes. I could write it as it is, the same way I write my name in the author's field. However, if a title, or any value in general, contains a colon, you do need to put the value in quotes. Here, I use the quotes to show that as an example.

You will notice that for the `author:` field I have a hyphen before my name. I didn't have to put that there either, but I did to show you a list. When you want a key to refer to a sequence of values, for example, if

you have more than one author on a document, you use hyphens before each element in the list. Here, I make `author` refer to a list of length one. The result is the same as if I hadn't put my name in a list, but if I had a coauthor, we would need the list syntax.

I have this header in a file called `header.yml`, and I can compile the book into a PDF file with the command

```
pandoc -o book.pdf header.yml book.md
```

The actual command line is more complex (see the Makefile in Chapter 5), but this command would suffice to generate a book.

The YAML language is essentially a way of mapping keys to values. In the preceding header, you have three keys: `title`, `author`, and `year`. A key can map to a single scalar value. `title` and `year` do that. They can also map to a list, as `author` does. Keys can also map to nested key-value mappings. Consider this:

```
author:
    - name: Thomas Mailund
      affiliation: Unseen University
    - name: Karsken Baelg
      affiliation: Brakebills University
```

Here `author` is a list—you can see this because you have dashes before each author. Each author is a nested mapping; they have two keys, `name` and `affiliation`. That it is another mapping is because they have keys followed by a colon and then values to the right of this. In short, scalars are keys followed by a single value, lists are keys followed by a sequence of items separated by hyphens, and nested maps are nested key-value maps. For lists and maps, there is a more concise notation. For a list you can put its values in square brackets and comma-separate them:

```
author_names: ["Thomas Mailund", "Karsken Baelg"]
```

For maps you can use curly brackets instead

```
author:
    - { name: "Thomas Mailund",
        affiliation: "Unseen University" }
    - { name: "Karsken Baelg",
        affiliation: "Brakebills University" }
```

Here, the items in the list have the same structure. They need not have this; it is easier to write code to process it when they do, though.

If you have a long text, you can break it into several lines in the YAML file using either | or >

```
abstract: |
    This is a very long abstract and
    it is probably the best paper ever.

    We are sure that you will all agree.
```

The difference between the two is that | will preserve linebreaks, while > will remove new lines and replace them with space. You can continue the text in these blocks as long as you want to, as long as you indent each line.

You can write arbitrarily complex YAML, but Pandoc will only use the metadata that it knows how to process. Which variables are interpreted by Pandoc depends on the output format and the template you that use (see Chapter 9). Check the Pandoc manual at https://tinyurl.com/yyxgole5 for details on the default templates, and see Chapter 9 for how to use metadata in your own templates.

CHAPTER 9

Using Templates

When Pandoc creates a standalone document, it uses a template for the output. A template is essentially a document with some placeholder variables, where metadata and your processed Markdown text will be inserted. Which metadata will be used in a template depends on the output format; you can get a full list of variables for your output in the Pandoc manual.[1] Pandoc automatically sets some metadata, described in the manual, but you can specify other metadata in the header.

Unless you specify another template explicitly, Pandoc will use a default for the output format. You can get Pandoc to show you the template it uses for a specific output by running the

```
pandoc -D <format>
```

command, for example, to see what it will use if you generate a PDF file—which it does by generating a LaTeX document and then compiling it—you can write

```
pandoc -D latex
```

You can also get a full list of default templates and what they look like at https://github.com/jgm/pandoc-templates.

If you don't know LaTeX, the template you get by running

```
pandoc -D latex
```

[1] http://pandoc.org/MANUAL.html#templates

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_9

might not make much sense, so let us look at

```
pandoc -D html
```

instead. The output is rather long, and I won't replicate it all here but highlight a few parts of it.

Remember the title we discussed in Chapter 8. It was empty before we provided metadata for the title. Let us see what it looks like in the template. There you will find a line that looks like this:

```
<title>
  $if(title-prefix)$
    $title-prefix$ -
  $endif$
  $pagetitle$
</title>
```

The stuff in dollar signs specifies placeholders and code for how the template should be processed. Inside the title tags in the HTML template, you have two metadata variables that can be inserted, `title-prefix` and `pagetitle`. The title prefix will only be inserted if it exists; that is what the `$if(title-prefix)$` code checks. Strictly speaking, `pagetitle` will also only be inserted if it exists. Otherwise, we get an empty string. But because the title prefix should be followed by a dash, there is an explicit test to see if anything should be inserted.

We never provided metadata for `title-prefix` and `pagetitle`, so it is hard to see how they relate to the `title` metadata we provided. We *could* have provided those two explicitly, but Pandoc creates them based on our title. It directly uses `title` variable elsewhere in the template, where the template contains:

```
<h1 class="title">$title$</h1>
```

Pandoc and filters can access metadata and create new metadata, which is what it does for the `title-prefix` and `pagetitle`. The `title` placeholder is just inserted directly as the text you specified in the metadata.

I am not aware of any documentation for precisely what metadata manipulations you can expect for each output format, and I am not sure you should rely on any as it might not be stable across different versions of Pandoc and filters. If you don't work with derived metadata and stick to explicitly defined metadata, however, how the data is used is relatively straightforward. If you have a simple placeholder like `$title$`, then the string you specified in the metadata will just be inserted there in the output file.

As we saw for `title-prefix`, metadata can also be inserted conditionally on it being defined. To insert `some text` only if a metavariable `variable` is defined, a template can contain this construction:

```
$if(variable)$ some text $endif$
```

There is also an if-else construction that looks like this:

```
$if(variable)$
some text
$else$
some other text
$endif$
```

Finally, there is a loop construction. In the HTML template, you can find this piece of text:

```
$for(author)$
<p class="author">$author$</p>
$endfor$
```

This runs through the authors specified in the metadata and inserts each of them. If `author` is not a list, it will still work; it will just be considered a list of length one, but if we did have a list of authors, we would get a level-two header for each of them.

Remember that metadata can be structured with values containing lists of key-value bindings. Take this example:

```
author:
- name: Thomas Mailund
  affiliation: Unseen University
- name: Karsken Baelg
  affiliation: Brakebills University
```

Here, authors are not simple strings, but a list of key-value structures, each with a name and an affiliation. Inside a template, these fields can be accessed using "dot-notation," so a template might contain code like this:

```
$for(author)$
 $if(author.name)$
   $author.name$
   $if(author.affiliation)$
     ($author.affiliation$)
   $endif$
 $else$
   $author$
 $endif$
$endfor$
```

This code iterates through the author list and inserts authors' names (if they have a name, which they probably should have). If they have an affiliation, the affiliation is added after the name. If the list of authors contains items that are not structured with a name and an affiliation, the template inserts the list item (see the `$else$` part of the `$if(author.name)$` test).

The most important part of the output, of course, is the processed Markdown from the input text. In the template, this is inserted at the hardly noticeable `$body$` placeholder. It doesn't look like much, but this is where all your Markdown will be inserted once it is processed to the output format.

Writing Your Own Templates

Templates are another of those features that are nice to have when you need them, but you don't have to worry about when you don't. You can use Pandoc without ever having to worry about templates, but if you have to format your documents in a specific way, you don't have to abandon Pandoc to write your text; you can create a template to take care of the formatting. For example, if you are an academic like me, and have to use different formats for different journals, you can make templates to match the journals. Journals often provide LaTeX templates for papers, and you can take one of those templates and put in Pandoc placeholders, and presto you have a Pandoc template and you can write the paper in Markdown and still have it formatted according to the journal standard.

You can get inspiration for writing your own templates from Pandoc's user-contributed templates.[2] I find that the easiest way to create a new template is to take one of Pandoc's existing templates and modify it or by taking an existing HTML or LaTeX file and put in metadata and `$body$` placeholders.

The reason that the default templates are lengthy and complicated is that they need to set up a long list of things, such as configuring math or source code highlighting. If you need all the features that Pandoc provides, I suggest that you copy one of the existing templates and modify it. For this chapter, I will write simpler templates, aiming for clarity rather than

[2] https://tinyurl.com/yyrgd66c

completeness. The Markdown input is correspondently simple. If you need more features, you can find the necessary code in the default template. It is usually not hard to find.

Template Examples

Consider this text. Most of it is metadata since that is what we are interested in here. As you can see, there are three variables in the YAML header: `title`, `subtitle`, and `author`. The first two holds a single value and the last is a list of simple values.

```
---
title:
  A terrible novel
subtitle:
  Seriously, one of the worst!
author:
  - Thomas Mailund
  - Karsken Baelg
---

It was a dark and stormy night
```

If we write the following HTML template, we add a title and a subtitle in the HTML header and the level-one header in the main document. If there are any authors in the metadata, then we insert a `div` element to right-align the list of authors, and we get the said list by iterating over them. The `$sep$` variable is not a metavariable as such but a way to tag the following token. If you use `$sep$`, then the token—word or comma, for example—will be put between all the elements you iterate over but will not follow the last element. Try removing it and you will see.

```
<html>
  <header>
    <title>
      $title$$if(subtitle)$: $subtitle$$endif$
    </title>
  </header>
  <body>
    <h1>
      $title$$if(subtitle)$: $subtitle$$endif$
    </h1>
    $if(author)$
      <div align="right">
        By
        $for(author)$$author$$sep$and $endfor$
      </div>
    $endif$

    $body$
  </body>
</html>
```

If we format our text with this template, we get the following HTML:

```
<html>
  <header>
    <title>
      A terrible novel: Seriously, one of the worst!
    </title>
  </header>
  <body>
    <h1>
      A terrible novel: Seriously, one of the worst!
    </h1>
```

```
        <div align="right">
        By
        Thomas Mailund and Karsken Baelg
      </div>

      <p>It was a dark and stormy night</p>
  </body>
</html>
```

For PDF/LaTeX output, this template does the same as the preceding HTML template.

```
\documentclass{book}
\usepackage{hyperref}

\title{$title$$if(subtitle)$: $subtitle$ $endif$}
\author{$for(author)$$author$$sep$and $endfor$}

\begin{document}
\maketitle

\end{document}
```

(The inclusion of `hyperref` is not relevant for the example, but it is one of those packages that Pandoc expects to be included. At the time I am writing this, it is the only one you need for this particular example).

Applying the template to our document gives us this:

```
\documentclass{book}
\usepackage{hyperref}

\title{A terrible novel: Seriously, one of the worst! }
\author{Thomas Mailund and Karsken Baelg}
```

```
\begin{document}
\maketitle

\end{document}
```

You can "dot" yourself into nested information in metadata, so if your data looks like this

```
---
title:
  A terrible novel
subtitle:
  Seriously, one of the worst!
author:
    - name: "Thomas Mailund"
      affiliation: "Unseen University"
    - name: "Karsken Baelg"
      affiliation: "Brakebills University"
---
```

and your template looks like this

```
\documentclass{book}
\usepackage{hyperref}

\title{$title$$if(subtitle)$: $subtitle$ $endif$}
\author{$for(author)$$author.name$
   from $author.affiliation$
   $sep$ and
   $endfor$}

\begin{document}
\maketitle
$body$

\end{document}
```

then progressing the document will generate this output:

```
\documentclass{book}
\usepackage{hyperref}

\title{A terrible novel: Seriously, one of the worst! }
\author{Thomas Mailund from Unseen University
        and Karsken Baelg from Brakebills University}

\begin{document}
\maketitle
It was a dark and stormy night

\end{document}
```

You need to get the name and affiliation for each author using `$author.name$` and `$author.affiliation$`. You cannot simply use `$author$` any more. You can still check if a value is set; just dot yourself into it. For example, if some authors do not have an affiliation, you can test for it and only insert it when it exists.

```
\documentclass{book}
\usepackage{hyperref}

\title{$title$$if(subtitle)$: $subtitle$ $endif$}
\author{
$for(author)$ $author.name$
  $if(author.affiliation)$
     from $author.affiliation$
  $endif$
  $sep$ and $endfor$
}
```

```
\begin{document}
\maketitle
$body$

\end{document}
```

If you have a LaTeX template as the preceding one (or for any other output format), you want it to apply to as many documents as possible. You don't want to have to update it every few documents you write. There are often a few tweaks necessary for each document, though. For example, in LaTeX you might need to import one special package or you want to define some commands. You don't want those modifications in your actual template, and you do not want to have many copies of the template lying around either. It is easy, however, to use metavariables to make your template adaptable.

Considering the case of LaTeX files as I just described, we can add packages and definitions to the template like this[3]:

```
\documentclass{book}
\usepackage{hyperref}

$for(packages)$
\usepackage{$packages$}
$endfor$

$if(definitions)$
  $definitions$
$endif$

\title{$title$$if(subtitle)$: $subtitle$ $endif$}
\author{
  $for(author)$
```

[3]Pandoc already has a metavariable, header-includes, for inserting LaTeX code into a template, but the example helps illustrate templates.

```
    $author.name$
    $if(author.affiliation)$
      from $author.affiliation$
    $endif$
    $sep$ and
  $endfor$
}

\begin{document}
\maketitle
$body$
\end{document}
```

We iterate over packages to insert each of them in a usepackage command. We just insert the definitions here. With this input

```
---
title:
  A terrible novel
subtitle:
  Seriously, one of the worst!
author:
  - name: "Thomas Mailund"
    affiliation: "Unseen University"
  - name: "Karsken Baelg"
    affiliation: "Brakebills University"
packages:
  - amssymb
  - amsmath
  - booktabs
  - xspace
definitions: |
```

```
  \newcommand{\TMRCA}%
    {\ensuremath{T_\textrm{MRCA}}\xspace}
  \newcommand{\tAC}%
    {\ensuremath{\tau_{AC}}\xspace}
  \newcommand{\tBC}%
    {\ensuremath{\tau_{BC}}\xspace}
  \newcommand{\tABC}%
    {\ensuremath{\tau_{ABC}}\xspace}
  \newcommand{\tadmix}%
    {\ensuremath{\tau_\text{admix}}\xspace}
---

It was a dark and stormy night
```

we get

```
\documentclass{book}
\usepackage{hyperref}

\usepackage{amssymb}
\usepackage{amsmath}
\usepackage{booktabs}
\usepackage{xspace}

  \newcommand{\TMRCA}%
    {\ensuremath{T_\textrm{MRCA}}\xspace}

  \newcommand{\tAC}%
    {\ensuremath{\tau_{AC}}\xspace}
  \newcommand{\tBC}%
    {\ensuremath{\tau_{BC}}\xspace}
  \newcommand{\tABC}%
    {\ensuremath{\tau_{ABC}}\xspace}
  \newcommand{\tadmix}%
    {\ensuremath{\tau_\text{admix}}\xspace}
```

```
\title{A terrible novel: Seriously, one of the worst! }
\author{
      Thomas Mailund
    from Unseen University
     and
      Karsken Baelg
    from Brakebills University
    }

\begin{document}
\maketitle
It was a dark and stormy night
\end{document}
```

If you do not want this LaTeX-specific code to turn up in HTML output, then simply do not include it in the template. If you want separate header configurations in HTML and PDF documents, you can use two different metavariables. For LaTeX macros (unlike the package inclusion), you do not need to modify a template. Pandoc understands the definition of macros and will apply a macro you invoke when producing output. If you call a macro inside math, Pandoc will produce the result of calling the macro in the math format the output needs. Outside of math, the result of calling the macro will be included in LaTeX and Markdown output but left out in other formats.

In case you are interested, the template I have used to format this book evolved over several books but currently looks like the following. I have not listed all the code that Pandoc has in its template but show, in a comment, where you can insert it.

```
\documentclass[11pt, openright,
               twoside, onecolumn, final]{memoir}
```

```
%% Setting up the paper size
\setstocksize{9in}{6in}
\settrimmedsize{\stockheight}{\stockwidth}{*}
\usepackage{canoniclayout}
\nonzeroparskip
\setlength{\parindent}{0pt}

%% Setting up the font
\usepackage[T1]{fontenc}
\usepackage{baskervillef}
\usepackage[scale=.95,type1]{cabin}
\usepackage[baskerville,vvarbb]{newtxmath}
\usepackage[cal=boondoxo]{mathalfa}

%% Disable hypertext (annoying in output)
\usepackage{nohyperref}
\usepackage{url}

%% Setup chapter heading
\renewcommand*\rmdefault{dayrom}
\chapterstyle{madsen}
%%%%%%%%%%%
%% A long list of commands taken from
%% the default template
%%%%%%%%%%%

\begin{document}

\frontmatter

%% Title page
\begingroup
\thispagestyle{empty}
{\bfseries\sffamily\noindent
```

```
$if(series)$ {\large $series$}\\[50pt]$endif$
% Book title
{\huge $title$}\\[35pt]
% Authors
{\Large $for(author)$$author$$sep$\\$endfor$}
\vfill
\endgroup

%% Copyright page
\newpage
~\vfill
\thispagestyle{empty}
% Book title
{\Large $title$}\\[15pt]
% Authors
\noindent Copyright
    $year$
    $for(author)$$author$$sep$, $endfor$\\
\clearpage
%% Table of contents
$if(toc)$
\setcounter{tocdepth}{1}
\pagestyle{empty}
\tableofcontents
\cleardoublepage
$endif$

%% Document body
\mainmatter
\counterwithout{figure}{chapter}
\pagestyle{plain}
```

```
$body$

\end{document}
```

The metavariables I have set for this book are

```
---
title:
  "The Beginner's Guide to Markdown and Pandoc"
author:
  - Thomas Mailund
year: 2019
---
```

There is no good reason that `author` is a list here. I have only coauthored one book, and it has been a list since, but for one author, it might as well have been a scalar value.

I admit that there is slightly more work involved with creating templates than just writing Markdown documents, but I do not believe that it is much harder to make a template than it would be to write the document in LaTeX or HTML in the first place. Plus, you have put all the complicated formatting in one document while you can focus on the content in another. If you reuse the same template multiple times, you amortize the time spent on creating it over many writing projects, and very quickly you will have saved time compared to writing in LaTeX or HTML directly in cases where you need more than one output format. And you can share your templates with friends and colleagues for gold or glory.

Exercises

Write your own templates; write one for HTML and one for LaTeX.

CHAPTER 10

Preprocessing

Markdown is just a plain text document, and you can do any rewriting of that text before you pass it through Pandoc. Any rewriting of the text before you give it to Pandoc is called *preprocessing*. Pandoc will read from standard input, so we can pipe the result of preprocessing into it on the command line (see Figure 10-1).

Assuming that the preprocessor takes the input file as input and that it writes its output to standard out, then a pipeline can look like this:

```
preprocessor infile.md | \
   pandoc --from markdown ... -o outfile
```

You need to tell Pandoc that it is getting Markdown as input if it reads it from standard in, and you do this with the `--from` option.

The preprocessor can do whatever you want it to as long as it outputs a file that Pandoc can process. The output does not need to be Markdown—you can change the `--from` option if it is not—but it must be a file in a format that Pandoc can read. I will use Markdown as my output in the following.

Figure 10-1. *Document formatting pipeline with a preprocessing step*

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_10

Examples

In the following examples, I will use GPP[1] for the first two and Python[2] for the last. GPP is a preprocessor with somewhat limited functionality, but for including files and for selectively including or excluding segments of a file, it works excellently. Getting Python to do the same is additional work. On the other hand, since Python is a general-purpose programming language, we can get it to do whatever we want with the input document.

Including Files

One use for a preprocessor is to have some information we can reuse in some files and another document-specific input—like a document's body—in another file. That is the idea with templates, but there are other cases we might have such a setup.

Imagine that you are teaching a class and hand out exercises every week. Some information, such as the name of the class and the name of the instructor, do not change from week to week but other information does, for example, the week number.

We can make a file `header.yml` with the general information

```
class: Markdown and Pandoc
instructor: Thomas Mailund
```

The header here is, of course, artificially simple. You only want to include a file that is of some complexity, but the example shows the principle.

For a specific week, we can then specify the week information, for example, the week number and the actual exercises for that week. Here is a file; let us call it `exercises.md`. It holds the exercises for week 14 of the class.

[1] https://logological.org/gpp
[2] https://www.python.org

```
---
#include "header.yml"
week: Week 14
---

# This is an exercise
Do something difficult

# This is another exercises
Do something even more difficult
```

The `#include "header.yml "` is where the preprocessor does its thing.

Notice that the three dashes delimiting the YAML specification are *not* in the `header.yml`. If it was then couldn't include it and still set the variable `week` in the `exercises.md` file. When we include it into the YAML header, we can combine the general variables set in `header.yml` with the file-specific variables.

If we pipe the document through the preprocessor

```
gpp < exercises.md
```

we get this result:

```
---
class: Markdown and Pandoc
instructor: Thomas Mailund
week: Week 14
---

# This is an exercise
Do something difficult

# This is another exercises
Do something even more difficult
```

We can combine this with a template:

```
\documentclass{article}
\usepackage{hyperref}

\title{$class$: $week$}
\author{$instructor$}

\begin{document}
\maketitle

\end{document}
```

Combining the preprocessor and Pandoc now lets us build a document with our exercises.

```
gpp exercises.md | \
    pandoc --template exercises.tex \
        --from markdown \
        -o exercises.pdf
```

Conditional Inclusion

Continuing with the exercise example, we could imagine that you have TAs for your class and you want to give them solutions to the exercise. It is easier to have the solutions in the same document as the exercises, but you don't want to hand the solutions to your student. So, what you want is a way to include the solutions when you make documents to the TAs and exclude them otherwise. This is something GPP is excellent at as well.

You can test if a variable is defined using `#ifdef`. A *variable* here should not be confused with the variables that Pandoc works with. Remember that the preprocessor sees the document before Pandoc and does not communicate with Pandoc other than piping its output into it.

If we want to include or exclude a block of text, we can put them between `#ifdef` and `#endif`. We can do that for the solutions to our exercises:

```
---
#include "header.yml"
week: Week 14
---

# This is an exercise
Do something difficult

#ifdef SOLUTIONS
This is the solution to the exercise
#endif

# This is another exercises
Do something even more difficult

#ifdef SOLUTIONS
This is the solution to the exercise
#endif
```

If you build a document as the preceding one, you will not get the solutions in the output. To get them, you need to define `SOLUTIONS`. You can do this in the file with a `#define` statement, but for this particular application, we might as well give them to `gpp` on the command line. Here we can use the option `-D`. This command line will build a PDF that contains both the exercises and the solutions.

```
gpp -DSOLUTIONS week14_exercises.md | \
    pandoc --template exercises.tex \
    --from markdown \
    -o week14_exercises_solutions.pdf
```

Running Code

Leaving the exercises, imagine that you are writing a book about programming and you have code examples. You want to show the result of running the code, so you want to evaluate all your code and insert the result into your document.

For example, you have the code

````
```python
for i in range(10):
    print(i, end = ' ')
```

```python
for i in range(10):
    print(-i, end = ' ')
```
````

and you want the first code block to be followed by the numbers 0–9 and the second from 0 to -9.[3]

This Python code iterates over all lines in the input. It uses `sys.stdin` to read the input, so you *must* pipe input to it and not call it with a file name. For each line, it checks if it is a code block line, that is, whether it starts with three backtics. If it is, and it starts with `python`, then it starts collecting lines until it sees the end of the block. When it gets there, it evaluates the python code, using `exec`. This function will execute the code producing any output the code prints—which is what we want here. Since we are using `exec`, functions and variables defined in earlier block scan be used in later blocks.

[3]In this book, whenever I present Python code, I assume that you use Python 3. If not, you need to adjust the code accordingly.

```
from sys import stdin

def main():
    exec_env = {}
    incode = False
    codeblock = []
    for line in stdin:
        print(line, end=“)
        if line.startswith("```python"):
            incode = True
            continue
        if incode:
            if line.startswith("```"):
                exec("".join(codeblock), exec_env)
                incode = False
                codeblock = []
                continue
            codeblock.append(line)

if __name__ == "__main__":
    main()
```

You can call the preprocess like this

```
python3 evalpy.py < eval-python.md
```

and get this result:

````
```python
for i in range(10):
    print(i, end = ' ')
```
````

```
0 1 2 3 4 5 6 7 8 9
```

```python
for i in range(10):
    print(-i, end = ' ')
```

```
0 -1 -2 -3 -4 -5 -6 -7 -8 -9
```

Exercises

If you have gpp installed, then preprocess a document such that you use a flag that gets you a different output when you create HTML and when you create LaTeX output. You have to explicitly set variables to do this, but see the next chapter for how to handle output formats in filters.

CHAPTER 11

Filters

Filters let you manipulate your documents similarly to preprocessors. Unlike preprocessors, they do not modify the text before Pandoc gets hold of it, but instead, they are plugged into the text transformation that Pandoc does. Think of them as post-processors; it is not far from the truth except that Pandoc will run after they are done.

Both preprocessors and filters have strength and weaknesses. They can do the same things to your files, but some things are easier to program in a preprocessor, while some things are easier to program in a filter.

Pandoc can read from standard input and output to standard output—it does this by default—and you can control the input and output formats using the `--from` and `--to` options. As with any pipeline, you can connect multiple programs, so instead of manipulating the input to Pandoc, as with a preprocessor, you can read the output from Pandoc, transform it, through as many steps as you like, and pipe it back into Pandoc for the final formatting (see Figure 11-1).

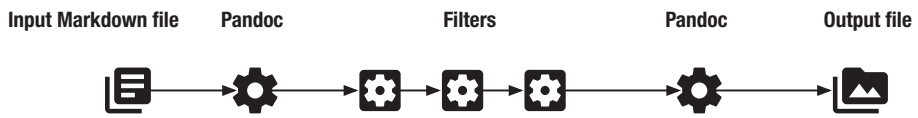

Figure 11-1. *Document formatting pipeline with filters*

You can string together any number of programs this way, as long as the output format of one matches the input format of the next, but what Pandoc thinks of as filters, and what you can add using the `--filter` or

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_11

`-F` options, should read and write in the JSON format. Consequently, any program that can read and write JSON can be used as a filter. You don't want to parse JSON yourself, though, if you can avoid it, so to write filters, you wish to use a software package/modules/libraries that help you rewrite the input.

There is support for filters in many languages, for some languages more than one package to support them; see a list at https://tinyurl.com/y2fsn89s. I am most familiar with Python, so in the following examples, I will use that language. The package I will use is `panflute` which is my favourite for writing Pandoc filters. To install `panflute`, you can run

```
$ pip3 install panflute
```

The JSON representation of a document can be thought of as a tree. A document contains paragraphs, paragraphs contain words and spaces, some words are emphasized, and so on. What the Pandoc filter packages typically will do is that they will traverse this tree structure and apply to each node a function that you provide. This function can leave the local tree structure alone, or it can return a modified tree. The output of the filter will be the input tree with your modifications.

You can use `pandoc <inputfile> --to json` to see the JSON representation of a file, but I find it easier to see the structure using `pandoc <inputfile> -- native`. The native format is a format used internally by Pandoc and is what you will traverse over with a filter. I suggest using it to recognize the structure a document will have, what to match on to recognize what you wish to rewrite, and what the rewritten structure should be.

Take a document like this:

```
# This is a level one header

This is a paragraph
```

The JSON format of this document is this:


```
{"blocks":[
    {"t":"Header",
        "c":[
            1,
            ["this-is-a-level-one-header",[],[]],
            [{"t":"Str","c":"This"},
                {"t":"Space"},
                {"t":"Str","c":"is"},
                {"t":"Space"},
                {"t":"Str","c":"a"},
                {"t":"Space"},
                {"t":"Str","c":"level"},
                {"t":"Space"},
                {"t":"Str","c":"one"},
                {"t":"Space"},
                {"t":"Str","c":"header"}
    ]]},
    {"t":"Para",
        "c":[{"t":"Str","c":"This"},
            {"t":"Space"},
            {"t":"Str","c":"is"},
            {"t":"Space"},
            {"t":"Str","c":"a"},
            {"t":"Space"},
            {"t":"Str","c":"paragraph"}]}
],
    "pandoc-api-version":[1,17,5,4],
    "meta":{}
}
```


It is a bit verbose which is why I prefer the native format:

```
Pandoc (Meta {unMeta = fromList []})
[Header 1 ("this-is-a-level-one-header",[],[])
    [Str "This",Space,
     Str "is",Space,
     Str "a",Space,
     Str "level",Space,
     Str "one",Space,
     Str "header"],
 Para
    [Str "This",Space,
     Str "is",Space,
     Str "a",Space,
     Str "paragraph"]
]
```

Except for the API version, which we are not concerned with for our filters, the two formats contain precisely the same information (obviously since it is JSON that is used between filters and a filter pipeline usually begins and ends with Pandoc).

The document has a metadata header (it is empty in this document) and then a list of document nodes. There are two top-level nodes, the header and the paragraph. The header has level 1 and then a triplet of extra information. The first element in the triplet is its identifier—it is used for hyperlinks when formatted as HTML and LaTeX. The next two elements in the triplet are classes and options. Inside the header is a list of strings and spaces. It is the text in the document. We will use identifiers, classes, and options in the following examples. The paragraph contains a lists of strings and spaces.

The hierarchy in this document is not deep. We have sequences of strings and spaces nested in the header and the paragraph. They can get deeper but usually not much. Consider this example:

```
---
oh: my
---

This is *very* important
```

If we get its structure, we get a string inside an emphasis inside a paragraph, but the depth is still small.

```
Pandoc
(Meta
  {unMeta = fromList
       [("oh",MetaInlines [Str "my"])]}
)
[Para
    [Str "This",Space,
     Str "is",Space,
     Emph [Str "very"],
     Space,
     Str "important"]
]
```

Here you also see an example of metadata. It looks more complicated than what we would expect from a simple map from `oh` to `my`, but it is what it is. In the following Python code, the `panflute` model will translate it into a simple map from keys to values.

To write a filter, try to make a small Markdown file containing input that you expect to transform and the output you want it to become. Then run the example through Pandoc to see what the native structure is. From that, and a package for Pandoc filters, you should be able to get what you want.

Exploring Panflute

Before we see concrete examples of filters, let us explore how `panflute` lets us traverse a document. As an example I will use this script:

```
import sys
from panflute import *

def print_structure(elem, doc):
    if type(elem) == Header:
        print("identifier:", elem.identifier,
              file = sys.stderr)
        print("classes:", elem.classes,
              file = sys.stderr)
        print("attributes:", elem.attributes,
              file = sys.stderr)

run_filter(print_structure)
```

The `run_filter` traverses the entire tree structure, depth-first, and calls a function we provide it; here that is `print_structure`. I have written a function that lets me show some of the properties of the header in the previous example. Let me add a few more properties to the header and process the document:

```
# This is my header {#header-id
    .class1 .class2
    foo=bar baz=qux}

This is *very* important
```

I have broken the header classes over multiple lines; Pandoc doesn't mind.

In the `print_structure` function, I ignore all document elements that are not `Header`—simply because I only do anything in case the element is a

header element. What I do is that I print the identifier, the classes, and the attributes of a header. I print them to standard error—if I printed them to standard out, I would mess up the JSON output that makes a filter work.

The text I print to standard error looks like this:

```
identifier: header-id
classes: ['class1', 'class2']
attributes: OrderedDict([('foo', 'bar'),
                         ('baz', 'qux')])
```

You can recognize the properties from the header.

```
# This is my header {#header-id
    .class1 .class2
    foo=bar baz=qux}
```

The identifier is first in the curly brackets, and it starts with `#`. The classes begin with a dot and otherwise standalone, and the attributes are key-value mappings. The different document elements have different properties but `panflute` is well documented, and you can find all the attributes each document element has.

The `print_structure` function doesn't explicitly return any values which means that it implicitly returns the `None` object. The `panflute` module will interpret that as saying that the function does not want to make any transformations but leave `elem` as it is. If we wanted to change anything, we must return an element to replace the input `elem`.

If you translate the preceding document into HTML, you can see how the various components are used. The identifier becomes the `id` in the header tag, the classes become classes, and the attributes become data attributes.

```
<h1 id="header-id"
    class="class1 class2"
    data-foo="bar"
```

```
    data-baz="qux">
  This is my header
</h1>
<p>This is <em>very</em> important</p>
```

Not all of the components are used in all output formats. In LaTeX, for example, only the identifier is used.

```
\hypertarget{header-id}{%
\section{This is my header}\label{header-id}}

This is \emph{very} important
```

That classes and attributes are not used much in the output here does mean that they are useless. You just need to find a use for them yourself. We can abuse them for our own nefarious purposes in our own filters.

Conditional Inclusion of Exercise Solutions

Consider this example from the previous chapter: we have text with exercises and solutions, and we want to compile it into documents where the solutions have been removed and documents where they have not. The preprocessing solution is excellent, but now we can see how we can achieve the same thing using a filter.

Let this be the input format:

```
Here is an exercise. Everyone can see it.

::: SOLUTION :::
Here is a solution to the exercise.
Do not give it to the students.
:::
```

The `::: SOLUTION :::` syntax is not one we have seen before because it is relatively rare. It creates a div tag in HTML—you can use it with a CSS file for formatting—and it adds a hyper reference target in LaTeX. We want neither of that, but we can use a Div structure in our filter.

You can see what the structure looks like by running this:

```
pandoc solutions.md -s --to native
```

where I assume that the Markdown is in the file `solutions.md`.

```
Pandoc (Meta {unMeta = fromList []})
[Para [
    Str "Here",Space,
    Str "is",Space,
    Str "an",Space,
    Str "exercise.",Space,
    Str "Everyone",Space,
    Str "can",Space,
    Str "see",Space,
    Str "it."
],
Div ("",["SOLUTION"],[])
 [Para [
    Str "Here",Space,
    Str "is",Space,
    Str "a",Space,
    Str "solution",Space,
    Str "to",Space,
    Str "the",Space,
    Str "exercise.",Space,
    Str "Do",Space,
    Str "not",Space,
    Str "give",Space,
```

```
    Str "it",Space,
    Str "to",SoftBreak,
    Str "the",Space,
    Str "students."]
 ]
]
```

The `::: SOLUTION :::` syntax sets the class of the Div structure to SOLUTION (the middle element in the Div object's properties). Classes are always lists, so it really sets the classes to a list with a single element, which is SOLUTION. If you want to give the Div object more attributes, you can use an alternative syntax:

```
::: SOLUTION :::
Solution 1
:::

::: {#solution-2 .SOLUTION .advanced}
Solution 2
:::

::: {#solution-3 .SOLUTION level=difficult }
Solution 3
:::
```

The first solution here has the same syntax as before. It will set the class of the Div object to SOLUTION and leave the other two properties empty. The second solution sets an identifier and adds another class to the list. The third solution is back to a single class but adds one key to value mapping to the attributes.

If you use this filter

```
import sys
from panflute import *
```

```
def print_structure(elem, doc):
    if type(elem) == Div:
        print("id:", elem.identifier,
              file = sys.stderr)
        print("classes:", elem.classes,
              file = sys.stderr)
        print("attributes:", elem.attributes,
              file = sys.stderr)

run_filter(print_structure)
```

on the preceding Markdown file, you will get this output:

```
id:
classes: ['SOLUTION']
attributes: OrderedDict()
id: solution-2
classes: ['SOLUTION', 'advanced']
attributes: OrderedDict()
id: solution-3
classes: ['SOLUTION']
attributes: OrderedDict([('level', 'difficult')])
```

We can use this information to process the example. We want to include or exclude solutions based on metadata. We could not easily do that in the preprocessor, but there we could use preprocessor variables which are harder to do here. Each method has its pros and cons.

The preceding information tells us that `'SOLUTION'` will be in the class of the Div if we have an `::: SOLUTION :::` block (or with the alternative syntax an `::: { .SOLUTION }" block`). We want to check the metadata to see if we should include the solutions. We can get the meta information from the `doc` parameter that `run_filter` will give our filter function. If we call `doc.get_metadata()`, we will get a table from which we can

get metavariables. We want our filter to remove Div objects that have SOLUTION as their class unless there is a metavariable called solutions and it is true.

This filter does that:

```
from panflute import *

def solution(elem, doc):
    if type(elem) == Div:

        if 'SOLUTION' not in elem.classes:
            # Return None to leave the node as it is
            return None

        meta = doc.get_metadata()
        if "solutions" not in meta:
            return Null
        if meta["solutions"] != True:
            return Null

        return None

run_filter(solution)
```

First, we check if `'SOLUTION'` is in the classes. If not, then we don't have a solution block and we leave the element alone by returning `None`. We could also have returned `elem`; it would make no difference. Otherwise, we get hold of the metadata. We check if `"solutions"` is in the metadata. If it is not in the metadata, then it definitely cannot be true, so we return `Null` as a replacement for `elem` in the output. Notice that it is `Null` and not `None`! The former is an element in Pandoc, while the latter is an element in Python; the former replaces `elem` with an empty block, effectively removing the solutions block, while the latter keeps `elem` as it is, that is, leaves the solutions block in the output. Finally, if `"solutions"` is in the metadata but not true, then we also remove the

solutions block. The `solutions` metavariable can have more than true as a value, and those would be considered as true as well in the expression `meta["solutions"]`, so I explicitly check for `True`.

If we do not get past the checks, we remove the solutions block. If the "solutions" is not set in the metadata, or if it is set to anything but `True`, we remove the block. Otherwise, we keep it by returning `None`.

If the block is not a solutions block, if `"solutions"` is not set or set to anything except true, then we have a solutions block, and we are supposed to keep it.

If we go back to the example Markdown

```
Here is an exercise. Everyone can see it.

::: SOLUTION :::
Here is a solution to the exercise.
Do not give it to the students.
:::
```

then we can check how Pandoc reacts when the metavariable `"solutions"` is set to false:

```
pandoc --metadata solutions:false -F solutions.py \
          solutions.md --to markdown
```

This is the result:

```
Here is an exercise. Everyone can see it.
```

The solutions block is removed in the output. The same happens if you do not set the metavariable or if you set it to any other value, except true. If you do set it to true

```
pandoc --metadata solutions:true -F solutions.py \
          solutions.md -- to markdown
```

the solution block is included.

```
Here is an exercise. Everyone can see it.

::: {.SOLUTION}
Here is a solution to the exercise.
Do not give it to the students.
:::
```

Pandoc uses this syntax in its output, but you can use either of the two ways to define a Div block.

Conditional Inclusions Based on Format

Sometimes we want a different text in the output conditional on the output format, for example, a different text for HTML and LaTeX.

You can insert raw text that is only included in the right output using text followed by `{=format}`. The delimiters for the text you write is slightly different based on whether you want a block or inline text. For a block, you write

````
```{=html}
See examples
<ul>
  <li><a href="#ex:ex1">Exercise 1</a></li>
  <li><a href="#ex:ex2">Exercise 2</a>
</ul>
```
````

For inline text, you use backticks:

```
See examples
`<a href="#ex:ex1">Exercise 1</a> and
 <a href="#ex:ex1">Exercise 2</a>`{=html}
 `\cite{ex:ex1} and \cite{ex:ex2}`{=latex}.
```

In either case, the quoted text is only inserted if the output format matches. We saw this syntax back in Chapter 6.

The text we include here will *not* be processed by Pandoc, so you cannot use Pandoc's features. That means that you cannot, for example, use the `[text](link)` syntax but must the hyperlinks in HTML. We will write a filter that allows us to do this.

We will use two or more classes. The first is used to tag that it is text that should only be output for some formats and another to indicate which output we want to output the text for.

We have already seen how to add classes to a `Div` block of text:

```
:::{.out .html}
This is only included for HTML
:::
:::{.out .latex}
This is only included for LaTeX
:::
:::{.out .html .latex}
This is only included for HTML and LaTeX
:::
```

For inline text, you have to use square brackets:

```
[HTML only]{.out .html} [LaTeX only]{.out .latex}
```

For blocks of text, we have to capture `Div` objects, and for an inline text, we need to catch `Span` objects. In either case, we need to check if we have the `out` class. Otherwise, we leave the object alone—it could be used for something else in another filter. If we have the `out` class, we check if the output format is also a class. If not, we remove the object; if the format is a class, then we include it.

The filter looks like this:

```
from panflute import *

def format_include(elem, doc):

    if type(elem) == Span:
        if not "out" in elem.classes:
            return elem
        if doc.format not in elem.classes:
            return []
        else:
            return elem.content.list

    if type(elem) == Div:
        if not "out" in elem.classes:
            return elem
        if doc.format not in elem.classes:
            return Null
        else:
            return elem.content.list

run_filter(format_include)
```

When we return the elements, we want to keep we do not just return `elem`. We don't necessarily want to have the `Span` and `Div` show up in the output. Instead, we get the object's contents. The script wants this as a list, so we use `elem.content.list`.

If you use the filter on this Markdown:

```
[HTML only]{.out .html} [LaTeX only]{.out .latex}

:::{.out .html}
This is only included for HTML
:::
:::{.out .latex}
```

```
This is only included for LaTeX
:::
:::{.out .html .latex}
This is only included for HTML and LaTeX
:::
```

you will get this HTML output

```
<p>HTML only </p>
<p>This is only included for HTML</p>

<p>This is only included for HTML and LaTeX</p>
```

and this LaTeX output

```
LaTeX only

This is only included for LaTeX

This is only included for HTML and LaTeX
```

There is no formatting here because there is no formatting in the input.

If you use another output format, for example, Markdown, then you will not get any output with this input; all the text is only included for HTML and LaTeX.

Evaluating Code

Now take another example we considered in the previous chapter: running Python code while formatting a document and inserting the results.

In this version, we will use classes to distinguish between code blocks we want to evaluate and those we do not. We only consider Python code blocks—those whose classes contain `"python"`—but to evaluate them, we will also require that they have the class `"eval"`. For example, in the markdown, before we will evaluate the second but not the first code block.

~~~{.python}
for i in range(10):
    print(i, end = ")
~~~

~~~{.python .eval}
for i in range(10):
    print(-i, end = ")
~~~

The filter is straightforward. Ignore for now the `run_python` we call—I list it in the following text, but it is not important how it works. Focus on `eval_python`. I realize that I have not been that inventive with the names, but `run_python_process` executes a Python process that evaluates a code block, while `eval_python` is the filter.

Consider the filter, `eval_python`. Here, we only look at elements of type CodeBlock. When we have a CodeBlock, we get hold of the classes and check that both `"python"` and `"eval"` are in it. If they are not, we do not enter the inner `if` statement, so the function will, by default, return `None` which leaves `elem` as it is. If we have the right classes, then we evaluate the Python code using `execute_code` function and insert the result after `elem` and return that.

```
import sys
from panflute import *

# definition of execute_code

def eval_python(elem, doc):
    if type(elem) == CodeBlock:
        classes = elem.classes
        code_body = elem.text
        if 'python' in classes and "eval" in classes:
```

```
            eval_res = execute_code(code_body)
            return [elem, CodeBlock(eval_res)]

run_filter(eval_python)
```

The execute_code function is slightly more complicated than the way we used exec in the preprocessor. It is simpler to use exec and let it print its output in the preprocessor compared to evaluating the code in a filter where we do not want any unwanted output.

If exec writes something to standard out, it will break the JSON format and this will break the rest of the pipeline.

Therefore, we need to capture the output of exec and then get hold of it again. Since the output of the code in exec gets sent to standard output, we need to change that into a file we can use, open that file when we execute code, close it again to flush it, open it, and read the result. It is not pretty, but it gets the job done, and you can do it like this:

```
PYTHON_IO_FILE = "/tmp/eval-python-io"
real_stdout = sys.stdout

exec_env = {}
def execute_code(code):
    f = open(PYTHON_IO_FILE, "w")
    sys.stdout = f
    exec(code, exec_env)
    sys.stdout.close()
    sys.stdout = real_stdout
    f = open(PYTHON_IO_FILE, "r")
    return f.read()
```

You do not need to understand this part of the filter to understand how the filter itself works.

We can run the filter and in this case get the result as Markdown:

```
pandoc -F eval-python.py eval-python.md --to markdown
```

The result is this:

```
``` {.python}
for i in range(10):
    print(i, end = ")
```

``` {.python .eval}
for i in range(10):
    print(-i, end = ")
```

    0-1-2-3-4-5-6-7-8-9
```

The output is not in a “backtick”-block but indented. This is just another way to write the same in Markdown.

If you use this document, you will see that we can define a function in one code block and use it in another

```
~~~{.python}
for i in range(10):
    print(i, end = ")
~~~

~~~{.python .eval}
print("defining foo")
def foo():
    for i in range(10):
        print(-i, end = ")
foo()
~~~

~~~{.python .eval}
print("calling foo from different block")
foo()
~~~
```

This is the output:

```
``` {.python}
for i in range(10):
    print(i, end = ")
```

``` {.python .eval}
print("defining foo")
def foo():
    for i in range(10):
        print(-i, end = ")
foo()
```

    defining foo
    0-1-2-3-4-5-6-7-8-9

``` {.python .eval}
print("calling foo from a different block")
foo()
```

    calling foo from a different block
    0-1-2-3-4-5-6-7-8-9
```

Numbering Exercises

As a final example, let us return to the exercise examples. This time, we are not concerned with including or excluding the solutions, but we want to put exercises in a LaTeX environment when the output is LaTeX and otherwise number them and add a header.

Consider an input like this:

```
::: Exercise :::
First exercise
:::

::: Exercise :::
Second exercise
:::
```

We have two Div blocks with class `Exercise` and some text within them. Those are the ones we want to modify. For HTML, say, we want to give them a header, and for LaTeX, we want to put them inside a LaTeX environment.

We can get the output format from the `doc` object. The input and output of filters are, as mentioned earlier, JSON, and if we just used shell pipes, we couldn't know what the final output is. When we run a script as a filter, however, Pandoc knows what the final output will be, and we can get that information.

The first attempt at the filter looks like this:

```
1  from panflute import *
2
3  no_exercise = 1
4
5  def number_exercises(elem, doc):
6      global no_exercise
7      if type(elem) == Div and \
8          "Exercise" in elem.classes:
9
10         meta = doc.get_metadata()
11
12         if doc.format == "latex":
13             exercise_env = "exercises"
```

```
14              if "exercise_env" in meta:
15                  exercise_env = meta["exercise_env"]
16              block = [
17                  RawBlock(r"\begin{" + exercise_env + "}",
18                           "latex"),
19                  elem,
20                  RawBlock(r"\end{" + exercise_env + "}",
21                           "latex")
22              ]
23              return block
24
25          level = 1
26          if "exercise_header_level" in meta:
27              level = int(meta["exercise_header_level"])
28
29          title = [Str("Exercise"),
30                   Space,
31                   Str(str(no_exercise))]
32          no_exercise += 1
33          return [Header(*title, level = level,
34                  classes = elem.classes), elem]
35
36  run_filter(number_exercises)
```

We use a global variable, `no_exercise` (line 3), for increasing the header number for each exercise. Inside the filter, we first check if we have a `Div` block with an `Exercise` class. If so, we get hold of the `meta` object from the `doc` element (line 10) and use it to check if its output format is LaTeX or HTML. If it is LaTeX (line 12), then we get the metavariable `exercise_env` (with `exercises` as default), and we create a new block as a replacement for the `Div` block.

The `RawBlock` is just verbatim text but only inserted if the output format is "latex." Of course, we know that the output is LaTeX here and we could leave out the argument, but in other cases, a `RawBlock` can be useful when you output roughly the same text for all output formats and do not want to check for the output format.

For all other formats (line 24 and below), we use a default level of 1 and otherwise use the metavariable `exercise_header_level` (lines 26 and 27). We create the header text (lines 29–31), increment the `no_exercise` variable (line 32), and then create the `Header` element. Its first argument is the list of text object that should comprise the header, then the header level, and keep the classes from the `Div` block. We put the `elem` text after the header.

Let us try it on HTML output (where the filter filename is `exercises.py` and the input Markdown is in `exercises.md`):

```
pandoc -F exercises.py exercises.md --to html
```

The output is this:

```
<h1 class="Exercise">Exercise 1</h1>
<div class="Exercise">
<p>First exercise</p>
</div>
<h1 class="Exercise">Exercise 2</h1>
<div class="Exercise">

<p>Second exercise</p>
</div>
```

As you can see, we have added a header to the exercises.

If we set the metavariable for the header level, we modify the level of the header:

```
pandoc --metadata=exercise_header_level=4 \
       -F exercises.py exercises.md --to html
```

```
<h4 class="Exercise">Exercise 1</h4>
<div class="Exercise">
<p>First exercise</p>
</div>
<h4 class="Exercise">Exercise 2</h4>
<div class="Exercise">
<p>Second exercise</p>
</div>
```

For LaTeX, we get this:

```
pandoc -F exercises.py exercises.md --to latex

\begin{exercises}

First exercise

\end{exercises}

\begin{exercises}

Second exercise

\end{exercises}
```

Here we do not create a header but put the exercises into an `exercises` environment. Since we do not add headers, the header level is ignored.

You need to define the environment in LaTeX for this to work. How to do this is beyond the scope of this book, but you can add an incantation like this in your YAML header:

```
header-includes: |
    \newcounter{exercounter}[section]
    \newcommand{\theexercise}%
    {\thesection.\arabic{exercounter}}
    \makeatletter
    \newenvironment{exercises}{%
```

```
    \par\refstepcounter{exercounter}%
    \protected@edef\@currentlabel{\theexercise}%
    \noindent\textbf{Exercise \theexercise}}{}
    \makeatother
```

and then have

```
$for(header-includes)$
$header-includes$
$endfor$
```

in your template before `\begin{document}`.

We can add references to the `Div` blocks.

```
::: {#ex1 .Exercise}
First exercise
:::

::: {#ex1 .Exercise}
Second exercise
:::
```

These are automatically kept for the `elem Div` blocks.

```
<h4 class="Exercise">Exercise 1</h4>
<div id="ex1" class="Exercise">
<p>First exercise</p>
</div>
<h4 class="Exercise">Exercise 2</h4>
<div id="ex1" class="Exercise">
<p>Second exercise</p>
</div>
```

You can now use the link syntax, text, to create hyperlinks to the exercises. If you want the reference to be in the header instead of the `Div` block, you can replace lines 33 and 34 with this:

```
identifier = elem.identifier
if not identifier:
    identifier = ""
header = Header(*title,
                identifier = identifier,
                level = level,
                classes = elem.classes)
elem.identifier = ""
return [header, elem]
```

It gives the header the elements identifier and sets the elements identifier to the empty string, which means that it will not be inserted in the output.

For LaTeX we want to use `\ref` commands; we do get hyper reference targets, but it is not what we want. We add a LaTeX reference command, however. To do this, we need to add a `\label` command inside the environments. Doing this is straightforward. Simply replace lines 11 to 22 with this:

```
if doc.format == "latex":
    exercise_env = "exercises"
    if "exercise_env" in meta:
        exercise_env = meta["exercise_env"]

    if elem.identifier:
        label = r"\label{" + elem.identifier + "}"
    else:
        label = ""
```

```
    block = [
        RawBlock(r"\begin{" + exercise_env + "}" +
                 label,
                 "latex"),
        elem,
        RawBlock(r"\end{" + exercise_env + "}",
                 "latex")
]
return block
```

We get the identifier for the `Div` argument—we have seen identifiers earlier—and then we insert it after the `\begin` command. If there is no identifier, we insert the empty string.

Now we have labels we can use with `\ref{}` commands in LaTeX (and we can insert those conditional on the output format) and we can insert links for other formats (dependent on those). LaTeX will automatically number the environments (if you have the LaTeX magic listed earlier to define the environment type), and it will automatically insert their reference number. For other formats, you need to insert the numbers yourself in the link.

This is not desirable. It means you have to manually update all numbers if you add an exercise inside your text. We need a better solution. We are going to use syntax similar to

`pandoc-crossref`

and

`pandoc-citeproc`

and we need to run our filter before `pandoc-citeproc` for the same reason that `pandoc-crossref` must. We don't want to interfere with references handled by these two filters, so we will give them a

prefix (like `pandoc-crossref`). Our references will look like this:

`[@ex:identifier].`

Before we can handle references, however, we want to collect a map from identifiers to exercise numbers. LaTeX will handle this for environments and `\ref{}` commands but for other formats we must. Since we are not guaranteed that we see an exercise before we reference it, we must traverse the entire document and make the map before we traverse it again and modify the document.

The easiest way to traverse the document is with the `run_filter` function, but we cannot run it more than once. There is another function, `run_filters`—notice the plural—that handles that. One filter will provide the input to the next; we don't want to modify anything with the filter that collects the map, so we just let it return None (implicitly by not returning another value).

The filter for making the map is straightforward and looks like this:

```
ex_dict = {}
no_exercise = 1

def collect_numbers(elem, doc):
    global no_exercise
    if type(elem) == Div and \
        "Exercise" in elem.classes:
        if elem.identifier:
            ex_dict[elem.identifier] = no_exercise
        no_exercise += 1
```

When we number the exercises, in the filter that modifies the document, we need to reset `no_exercise`. This is not easy when the two scripts are called one after another, but a straightforward solution is to use a second counter. If you do this, then the preceding `number_exercises`

filter will work as before. It adds the numbers, but it didn't need the map earlier, and it doesn't need it now.

Instead, we will write a third filter that does this; let us call it `handle_citations`. I describe this function in the following text.

We can call the three filters using

```
run_filters([
    collect_numbers,
    number_exercises,
    handle_citations
])
```

We must run `collect_numbers` before `handle_citations`, but `number_exercises` can go anywhere in the list.

If you get the `native` format for a file that contains these `[@ref]` references, you get a complex text, but you will see that we have a `Cite` object that contains a `Citation` element (there can be more than one, but we will only handle one here). We want to translate these elements. We really want to work with `Citation`, but if we filter on that, we will create an object that goes into the `Cite` that encloses it, so we will handle `Cite` objects and extract the `Citation` object from it.

A `Cite` object contains several attributes including the identifier (the `@reference` text), a prefix (text that goes inside the square brackets but before the reference), and a suffix (text that goes after the text). We will only use the prefix and the identifier here and ignore—effectively remove—the suffix. See the exercises for including the suffix.

```
1   def handle_citations(elem, doc):
2       if type(elem) == Cite:
3           actual_cite = elem.citations[0]
4           identifier = actual_cite.id
5           if not identifier.startswith("ex:"):
6               return elem
7
```

```
 8            prefix_text = actual_cite.prefix.list
 9            prefix_text.extend([Space, Str("exercise")])
10
11            if doc.format == "latex":
12                return actual_cite.prefix.list + [
13                    RawInline(r"~\ref{" + identifier + "}",
14                              "latex")
15                ]
16
17            if identifier in ex_dict:
18                ex_num = ex_dict[identifier]
19                prefix_text.extend([
20                    Space, Str(str(ex_num))
21                ])
22            return [
23                Link(*actual_cite.prefix.list,
24                url = "#" + identifier)
25            ]
```

In the first line in the filter, we check if we have a `Cite` element. There is nothing new there. Then we extract the element we are actually interested in, which is a `Citation` element.

We get the identifier, which is the citation label (line 4), and check if it is an example label, that is, starts with `"ex:"` (line 5). If it is not, we return the element; we do not want to modify other citation objects since these could be used by other filters.

Now we extract the prefix of the `Citation` object; again we get the actual content using `.list`. We add the text `"exercise"` to the prefix (line 9), so the references will contain this text as well. We need to add a space as well to prevent the prefix from being concatenated with the `"exercise"` text. We also need to add a space after `"exercise"`, but I

want a non-breaking space in the LaTeX output, so there I want a tilde rather than a space (see the following text).

I am assuming that there is already a space before the reference, that is, that the reference looks like this `[see @ex:ref]` rather than `[see@ex:ref]`; if not we need to add a space before `"exercise"`. We will need to add it after the reference. Otherwise, the prefix and the reference number will be concatenated.

We now handle LaTeX output separately (lines 11 to 15). We append the `\ref{}` command to the prefix and return the result. We put the LaTeX code in a `RawInline` object. This is similar to a `RawBlock` object but for inline text. We do not add a space here but a tilde.

For other output formats, lines 17 to 25, we look up the exercise number from our map, assuming that there is an identifier, and then we add a space and the number to the prefix. We add a space before the number, so it isn't concatenated to `"exercise"`. Finally, we create a link from the reference.

Exercises

Conditional Inclusion

Modify the filter so you can use a metavariable to determine which difficulties should be included and which should be removed.

Conditional on Output

We removed the `Span` and `Div` objects when we modified the input. If the objects had more than the output and format classes, we might want to keep them (but with the `.out` and `.format` classes removed). Modify the script to do this.

Evaluating Code

Add a class to the code blocks that will determine whether the original code and the block should remain in the output, while you still evaluate the code in the block.

Numbering Exercises

Can you add the reference suffixes to the output as well?

CHAPTER 12

Conclusions

By now, you have seen most of the features of Markdown and Pandoc. I have not covered all the features, but you should have a good idea of what you can do with these tools and be able to learn more from online manuals.

With Markdown you do not have quite as much control over typesetting and document structure as you would have, for example, in LaTeX, but the substantially simpler syntax for many markup instructions makes it much easier to work with. Especially for tables, lists, and figures, where LaTeX's syntax can take the focus away from the actual content of your document.

From time to time, you need more than Markdown can do by itself, but then Pandoc has several handles you can turn. If you need to specify formatting beyond Markdown, you have templates, and if you need to transform your document while formatting it, you can preprocess it or use filters to rewrite it.

If you have to write new templates and new filters for each new document, then there is nothing gained from using Markdown and Pandoc compared to formatting each document manually, using, for example, LaTeX or Word. If you are like me, however, you can reuse a few templates for all your documents, and the occasions where you need a new filter are few and far between—and I have never experienced writing a filter that I did not use more than once.

I hope that you have found this introduction to Markdown and Pandoc instructive and that you will enjoy writing Markdown in the future.

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2_12

Index

T. Mailund, *Introducing Markdown and Pandoc*,
https://doi.org/10.1007/978-1-4842-5149-2

F, G

H

I

J, K

L

M

N, O

P, Q

R

S

T

U

V

W, X

Y, Z