目　录

致谢

README

This is a book on constructing ebooks using Markdown. No esoteric and complicated formats. No WYSIWYG editors. No PDF hell. Just plaintext.

You’ll learn everything from generating a TOC to adding cover images.

It’ll teach you the very workflow I used to write this book. A workflow that frees you from the constraints
of writing in a complicated format that scarcely resembles text, or writing

You’ll learn the following;

- The differences between various Markdown implementations
- How to take Markdown and turn it into;
  - epub
  - mobi
  - pdf
  - html
- How to work with code samples, codeblocks, images, TOC etc
- How to build custom Markdown extensions using Kramdown. For stuff like;
  - code bubbles
  - notes
  - asides
  - and other nifty little doodads
- How various html ebook format wrappers (such as ePub, mobi etc) work. No magic!
- Common markdown pitfalls (I documented all my failures while writing this book)
- Various techniques for previewing your Markdown output
  - Sublime Text Previewing
  - html previews

  - automatic previews using the guard gem
- How to mirror the Leanpub workflow using local tools
- How to customize converters so you have full control of your in-between formats; laTeX (pdf) and HTML (ePub, mobi etc)
- The minimal laTeX you'll need to get stuff done with PDF

What you need to know already

Some hacker skills are necessary to make full use of this book. To write books using Markdown you'll need to be comfortable getting your hands dirty, breaking things and learning. This book doesn't assume you're skilled in any particular language but it assumes you are skilled in programming. Familiarity with your systems package manager will be necessary.

Can you do this?
$ homebrew install

Or if you are on Linux this?
$ sudo apt-get install

If this leaves you scratching your head then you are not ready for this book. If you are left feeling comfortable and at ease, not queasy and fearful, then you are ready.

Some rudimentary ability to read and edit Ruby will be necessary. If you are comfortable enough to edit Gemfiles and
install gems using bundler then you should be fine.

Table of Contents

Planned Chapters

Chapters on the following topics are planned;

- Using Quarto and PDFKit to generate PDFs from XHTML + CSS
- Web based books with runnable consoles and code samples

This book is currently in beta

It is considered to be about 90% complete and is still undergoing proofreading, editing and revision.
The last 10% can make a big difference in the quality of a work so please be forgiving if it does not
meet your expectations.

A quick note on licensing

This book is licensed under a MIT license and is open sourced on Github.
This does not mean I want you to read it for free, in fact, I want it to be worth more to you. I recognize that the
more permissively licensed the more valuable it becomes. The more free my work is to move the greater the potential
impact it can have.

My goal is to make the book as valuable to you as possible. So the less I limit you the better.

Feel free to take my book and use it in whatever way you like, in whatever way furthers your life the most. I only ask
that in return, you contribute back to me what I contributed to you.

How much this work is worth to you is up for you to decide. I hope you find it's a lot.

You can purchase this book on Leanpub or if you like you can just drop me some
bitcoin: `1csGsaDCFLRPPqugYjX93PEzaStuqXVMu` or support me on Gittip

~K-2052

来源(书栈小编注)

> *https://github.com/k2052/markdown-to-ebook*

空标题文档

```
These are templates that get passed to Kramdown.
```

An Introduction

An Introduction

Writing a book can be daunting, learning the tools to write a book can be even more daunting. A quick google search will turn up a multitude of services, tools, apps and frameworks designed to write a book. Apps like; iBooks, Adobe Acrobat, inDesign Skyreader, Kindle Publisher etc. Formats like; PDF, ePUB, LaTeX….

It’s not so far fetched that a writer when faced with this mess of “solutions” might simply choose a trusty old
typewriter or even pen and paper. Or he might get so caught up writing tools for writing that he never gets the time to
write.

This book is a result of my search for a get out of the way method of writing ebooks. Something that keeps my focus on
the writing and not on the process of writing. It doesn’t cover everything, it gives you just enough knowledge so you
can focus on what matters, the writing.

This book follows the hackers ethos of not obscuring things. Problem areas are not glossed over, they are put right out
in the open, so you can learn from them. Where I struggled you will know I did and why I did. I wrote this book through
countless frustrations with Markdown parsers, LaTeX packages and varying bugs; to ignore that would skip a crucial part
of how I learned what I know.

Most technical books appear smooth when read, but they were not written smoothly. To get that snippet down to something
slim and slick requires a lot of breaking things and figuring out what doesn’t work. Making things look easy is hard
work. I tried not to make things look easier than they are. By showing you my failures I hope to help you avoid
them.

This is not a book for those that like quick summaries and step by step

processes. It’s a book for those that like to
learn something. The only way to eliminate frustration is to internalize
your knowledge, to make it so much a part of
you that it cant be ignored.

Unless you truly know how to do something then doing it is going to get in
the way and take energy away from your
writing. You cant for instance, be looking up how to create a section
every time you want to create a section. Such a
workflow will kill a book before it is even start. Things have to flow
smoothly, without thought, so you can just write.

With this in mind, I have kept the approach to tools as simple as
possible, so they can be internalized as quickly as
possible. Our book’ source files will be written in Markdown. Markdown is
simple enough to internalize in a day and
most importantly, it is flexible enough that it can be adapted to support
a variety of extras.

Kramdown has been chosen as our flavor/implementation of Markdown. This is
for a few reasons;

1. It is used in a variety of publishing platforms (specifically Leanpub)
2. It closely resembles many other implementations like PHP Markdown Extended.
3. It’s well documented.
4. It’s coded in Ruby. Tools like pandoc are just as powerful as Kramdown but pandoc is unfortunately written in
   Haskell. Haskell would only increase the learning curve and add more annoying parts to getting our
   ebook published.

What you need to know

You wont need to know much to get through this book but you will need to
be comfortable hacking around. Comfortable is
a dangerous word for communication, one’s interpretation can vary greatly
from what is intended. Let’s go over some
examples to help clarify what I mean by *comfortable*

Can you do this?

{:lang=”sh”}

```
homebrew install
```

Or if you are on Linux this?

{:lang=”sh”}
sudo apt-get install

If this leaves you scratching your head then you are not ready for this book. If you are left feeling comfortable and
at ease, not queasy and fearful, then you are ready.

Some rudimentary ability to read and edit Ruby will also be necessary. If you are comfortable enough to edit Gemfiles
and install gems using bundler then you should be fine.

We are going to need a workflow

We are going to need a workflow

I’m pulled in a thousand different directions by a thousand different solutions. iBooks Author looks nice. Nathan Barry went with that right? I think that is also how Josh Long wrote Execute. But what about epub? Isn’t epub just html at its core? Maybe I’ll write in HTML and use an HTML to PDF converter? That would allow me to craft all sorts of customizations in CSS. I could really style things awesomely. I could release a web version with fancy JS stuff and blow people’s minds with a really unique ebook release.

But what about organization of my writings — writing stuff in pure HTML is going to be a mess. I guess I could use a static site generator and write in a templating language to keeps things organized into chapters. But maybe I should use a ebook authoring tool that handles this for me? What about Scrivener? Don’t screenplay authors use that? What if I want to deliver a screenplay for my book in case they decide to adapt into a movie?

Choosing a workflow from a place of ignorance is an exercise in futility. Any choice we make will be clouded by a lack of knowledge, too many options and possibly delusions of grandeur.

If we want to find a workflow that works we are going to have start with very basic questions. Questions we can see the answers to immediately. If we answer the simple questions first then the complex ones will become clearer. Answering some basic questions will give us constraints and illuminate the path ahead.

Where will we distribute the book?

For distribution it would be nice to have something that works well for the writing phase as well as the publishing
phase. Perhaps even something that generates our ebooks from our source formats for us. Such a service exists and it’s
called Leanpub.

What does Leanpub use as its source format? Leanpub uses Markdown. Thus, we will need to use Markdown; we now have our
first constraint.

What final format does it need to be in when distributed?

There are a zillion different options for ebook formats so whatever we choose it’s going to have to be capable of
converting into a variety of formats. We will need something that can go into PDF, epub, kindle and mobi formats. It
just so happens that with the exception of PDF all these formats are HTML at there core.

What is a good option for writing HTML ebooks? Markdown of course. Markdown is simple, flexible, looks great as
plaintext and most importantly, it is built with HTML as the intended output format.

How then do we turn Markdown into PDFs? The answer lies in LaTeX, which will be our in-between format of choice. LaTeX
is to PDF as Markdown is to HTML. They’re best buds and get together all the time to just chat and mark things up.

Does LaTeX mean we will have to maintain two separate formats? Nope. Markdown can be easily converted into LaTeX
using tools like Kramdown.

LaTeX

LaTeX is one of those things you probably know about but have never bothered learning. I mean, it’s for technical

papers right? When have you ever needed to write a technical paper. One day when you have the time, you plan to study
it but not now. For now, you’ll just use Markdown. Markdown works for now, but one day you’re going to have to touch
LaTeX if you want to move beyond the HTML realm.

I could cover LaTeX the normal way; mention the apps you use to use it, the commands you use to install it, the syntax
you use to craft it, and then send you on your way. Most books on LaTeX are written this way. I could tread the same
ground other books do and I’m sure you’d get some benefit. After all, countless books cover things that way, it’s not a
completely useless approach.

You might even be motivated enough to play with LaTeX a bit. Perhaps you will even write a few a things in it. In the
long run though, you’ll slowly find its usefulness lacking. Markdown will just be quicker and more productive. When you
write Markdown you’ll get things done. When you use LaTeX you will feel like you’re fighting it; spending more time on
your tool than on writing. When you truly need the power of LaTeX you’ll do what you always do, live without it or hack
a solution onto your favorite Markdown parser.

I know you’ll do this because I do it. We all do it. We learn new tech because we are curious, but if the logistics
don’t work out, if the practicalities of using it don’t align with execution, actually getting stuff done, then we
don’t use it. It’s the same reason we never blog on our amazing overly designed custom built blogging engines, it just
isn’t productive. One day a Medium, Tumblr or Jekyll comes along and we finally take the plunge and simplify enough to
just focus on our writing. Things need to be too easy or they won’t be made use of. Your tools need to blend into the
background and get out of the way of creation or you’ll never use them.

If we are actually going to make real use of LaTeX it’s going to have to work for us, not us for it. It will have to
complement our workflow, not supplement it. We will adopt a workflow that makes LaTeX work for us.

Leanpub

Let’s visit Leanpub, get registered, and create a test book.

After we get signed up, Leanpub will inform of us the basics. We will upload our ebook files via Dropbox. They will be
written in plain text using Markdown.

We will get a link sent to us for our first book. Get Dropbox setup if you haven’t already.

I> You can utilize the web interface for Dropbox. No need to download/install.

Visit the getting started page for your book *leanpub.com/bookname/getting_started* e.g *https://leanpub.com/markdown-to-ebook/getting_started*

The first thing we can take away from this page is that we will use Markdown. If we incorporate fancy stuff it
will be by customizing the conversion of our Markdown. I actually like this as it keeps our text straightforward and
simple, we will only have to use convoluted LaTeX when absolutely necessary.

We need to know a little about the Markdown flavor we are dealing with though. After some quick googling I found this
HN comment:

```
Hey, co-founder of Leanpub here.

If you're interested in the differences between Pandoc and Leanpub, there are two places to look. Leanpub is based
off of Kramdown, so there's the Kramdown documentation[1]. We have made a few extensions to Kramdown to support
things that books need, so you'll also want to look at the Leanpub manual[2].

If you have any questions, send us an email at hello@leanpub.com.

[1]: http://kramdown.rubyforge.org/syntax.html [2]: https://leanpub.com/help/manual
```

So, Leanpub uses Kramdown, this is good news. This makes incorporating Leanpub into our workflow even easier.

Flavors of Markdown

Flavors of Markdown

You might think you know Markdown, but chances are you only know *a Markdown*. The Markdown you know might not be the
Markdown you get to use. All Markdown implementations handle the basics but you might not be aware of what the basics
[^MarkdownExtensionsSpec] actually are. Automatic link parsing, for example, is not part of the standard, it’s just
something that everyone and their biological birthing entity implements. There are three major flavors of extended
Markdown you are likely to encounter.

[^MarkdownExtensionsSpec]:
An official extension spec is being worked on but the details are pretty sparse. See
The Future of Markdown
and this commit

Github flavored Markdown

This implementation has been made enormously popular by the ubiquity of Github in the development community. It
has a few features not found in the original Markdown spec, namely; fenced code blocks and automatic link parsing. The
de facto implementation of this flavor is redcarpet

MultiMarkdown

Like many Markdown flavors, MultiMarkdown originated as the de facto implementation of Markdown for a
language; originally written for Perl, MultiMarkdown has since become a C

library. It is one of the
most well documented flavors but it lacks a decent adoption rate/userbase.

Markdown Extra

This variant came originally from PHP but has since been implemented
in several languages. Markdown Extra has a few extra features like;
footnotes, extensions, attribute lists etc.

We will be using an implementation of Markdown Extra called Kramdown.
There are two reasons for this:

1. Leanpub uses it.
2. It is written in Ruby which is easy to hack on.

For the most part, the only two flavors you will have to deal with are
Github’s and Markdown Extra. The differences
between the two aren’t too significant. Most of what Github does is extra
for Github and won’t effect your needs for
ebook writing. Utilizing Github features just adds stuff when your
Markdown is used on Github and utilizing Kramdown
features just adds stuff when using Kramdown.

The only difference you need to care about is the syntax for fenced code
blocks. Github utilizes three `‘s and Kramdown uses
three `~~~`.

They look like:

Github

~~~
```ruby
require 'redcarpet'
markdown = Redcarpet.new("Hello World!")
puts markdown.to_html
```
~~~

Kramdown

```
~~~ ruby
require 'redcarpet'
markdown = Redcarpet.new("Hello World!")
puts markdown.to_html
~~~
```

Since you cant modify the parsing on Github, if you plan to publish the same Markdown on Github and need the code
highlighting you'll need to modify Kramdown to use the same indicator as Github for starting and closing fenced code
blocks.

A> Alternatively you can just use indentation to specify your code blocks which is compatible across most
A> implementations

Changing `~` to a ` will get it parsing like Github. To do that just search for `FENCED_CODEBLOCK_START` in the
Kramdown files. Monkey-patching is the simplest way to get the modification into Kramdown. In the
interest of clarity, the following code snippet will accomplish said changes:

{.lang="ruby"}
module Kramdown
module Parser
class Kramdown
FENCED_CODEBLOCK_START = /^ `{3,}/ FENCED_CODEBLOCK_MATCH = /^(` {3,})\s?(\w+)?\s?
\n(.?)^\1`\s*?\n/m
end
end
end

If you plan to use Leanpub you'll need to utilize indented codeblocks instead. You'll indent things using four
spaces then manually specify your language using attribute lists. Like this:

{:lang="md"}
This is a codeblock

```
{:lang="ruby"}
    cats = 'demons'
```

Beyond that, just follow the docs for Kramdown and you'll be fine.

Learn your Markdown

You might be tempted to avoid learning Markdown and to add a Markdown GUI to your workflow. I urge you to resist that
temptation. Jumping around in app menus for simple things like links and headings will kill your workflow. There is a
reason why some writers from past eras still choose to use typewriters, being 100% comfortable with your tools is
extremely important for productivity. You'll need to know your Markdown inside and out if you ever hope to finish a
book.

Any app that attempts to replace the Markdown text process is going to interfere with your workflow in the long run. If
you use Markdown apps only use them to complement your writing, with fancy previews and nifty publishing tools. Don't
use an app as a crutch. Using crutches will make you disabled.

Leanpubbin’: A Brief Introduction To Leanpub

Leanpubbin’: A Brief Introduction To Leanpub

The best way to get started with Leanpub is just to dive in — into a sample book. Visit https://leanpub.com/books/new and follow the steps.

Leanpub will have created a folder for your ebook on Dropbox. Open up the folder. You should see *convert_html* and *manuscript* folders.

Open up the manuscript folder. You will see a Book.txt file and a chapter1.txt file.

Book.txt is an index of files. Files are ignored (not included in the book) until they’re added to the index. Leanpub
takes Book.txt, parses it line by line, joins each file and then parses the joined file with Kramdown. At some point in
the future Leanpub takes the Kramdown file and converts it to pdf, epub and mobi formats.

If we wanted to create the structure for a Leanpub book manually we could do the following:

Create the manuscript folder:

{:lang=”sh”}
$ mkdir manuscript

Go into the folder:

{:lang=”sh”}
$ cd manuscript

Create a Book.txt file:

{:lang=”sh”}
$ touch Book.txt

Now we add chapters to our index:

```
001_Introduction.md
```

and so on..

When formated as Markdown a Leanpub book looks like:

{:lang=”md”}

```
# Chapter Title

Chapter contents

## Section Title

Section contents
```

Leanpub Markdown flavor

Leanpub Markdown is based on Kramdown. The main thing you need to know beyond
Kramdown’s documentation is:

1. Use h1’s for chapter titles.
2. Use h2’s for section titles.

The h1’s and h2’s will be parsed and turned into your Table of Contents, so it’s important you don’t utilize them for
things other than chapter titles and section titles.

Images in Leanpub

All assets are linked relative to your manuscript folder. Images are placed in *manuscript/images*. Link to them using
the normal Markdown image syntax.

For example;

{:lang=”md”}

They need to be at 300ppi if you intend to distribute you book via print. They will be upscaled to 300ppi automatically
so if you intend to distribute entirely through digital means you can forgo creating them at 300ppi. It might be a good
idea to create them at 300ppi anyway just to keep your options open.

If you need your images in 300ppi (e.g after taking a screenshot) the easiest thing to do is run an imagemagick command
against it:

{:lang=”sh”}
$ convert -units PixelsPerInch image -density 300 outimage

Setting the density will be adequate if all you intend to do is distribute your book digitally. If you do print you’ll
have to make sure to take them at retina resolutions and then do some post re-sampling. Something like:

{:lang=”sh”}
$ convert -units PixelsPerInch image -resample 300 outimage

You can pass attributes like width and height they will be reflected in the final output. For
example:

{:lang=”md”}
{width=60%,float=left}

```
![This is the Image Floated Left](images/LeanpubLogo1200x610_300ppi.png)
```

I> See the how-to-insert-an-image part of the Leanpub manual
I> for more details.

However, I recommend you don’t take advantage of Leanpub’s image attributes. They are not parsed like Kramdown’s and
their syntaxes are incompatible. If you use Leanpub’s syntax will result in stray text (e.g `{width=60%}` _ appearing
when you use Kramdown to generate PDFs. The best solution, is to simply not rely on image attributes.

If you absolutely need image control in the final published PDF then use

attribute lists, which are compatible with
Kramdown.

The Kramdown syntax looks like this:

{:lang=”md”}

{:width=100%}

Previewing Leanpub locally

Previewing is central to a Leanpub workflow, you’ll be doing it a lot. So much, that manually previewing is likely to
get cumbersome. To keep your sanity you are going to need a variety of methods to quickly preview changes. Methods that
don’t require you to wait for Leanpub to process your book.

T> Previews from Leanpub will be generated into your Dropbox folder in */Preview*. You can manually edit these if you
T> want to get clever but I highly recommend you only use them for previewing.

Previewing with Sublime

Thankfully someone has built an extension for this called sublimetext-markdown-preview. We can install it using Package Manger;

1. *Shift + CRTL + P*.
2. Install Package: *Markdown Preview*.

Once you have it installed you can just hit *CRTL + SHFIT + P* (P is for preview) and a browser window will open
with a preview.

Previewing with VIM

There are a few scripts for this. Here is one

Previewing with Guard and HTML

A good cross-platform, cross-editor, solution, is to monitor your Markdown files and generate HTML when they change,

then you can just use your browser to preview the result. This solution has the added benefit of allowing us to
generate previews using Kramdown, which means our previews will support everything Leanpub does.

To do this we will use guard and a guard script that monitors Markdown files.

First we have to install Guard. The process is fairly simple:

{:lang=”sh”}
$ gem install guard

A> Add `sudo` if installing to system. Drop the `sudo` on Mac. Also drop sudo when using rbenv. On windows, drop
A> everything, run around screaming, then install a decent Operating System.

In your ebook folder run:

{:lang=”sh”}
$ guard init

Now install the markdown monitor:

{:lang=”sh”}
$ gem install guard-markdown

Add an example guard for Markdown to your Guardfile by running:

{:lang=”sh”}
$ guard init markdown

It will generate something like this:

{:lang=”ruby”}
guard ‘markdown’, :convert_on_start => true, :dry_run => true do
watch (/source_dir\/(.+\/)*(.+.)(md|markdown)/i) { |m| “source_dir/#{m[1]}#{m[2]}#{m[3]}|output_dir/#{m[1]}#{m[2]}html”}
end

We want to change the source dir to `/manuscript, set the output_dir, and enable it:

{:lang=”ruby”}
guard ‘markdown’, :convert_on_start => true do

watch (/manuscript\/(.+\/)*(.+.)(md|markdown)/i) { |m| “manuscript/#{m[1]}#{m[2]}#{m[3]}|preview_html/#{m[1]}#{m[2]}html”}
end

Launch it with:

{:lang=”sh”}
$ guard

A> Run this in the ebook’s folder

Now we need to make sure our browser gets reloaded when the files change. We can do this with *guard-livereload* and a
LiveReload browser extension. Once you have the relevant browser extension installed do the
following:

{:lang=”sh”}
$ gem install guard-livereload

Then add the following to your Guardfile:

{:lang=”ruby”}
guard ‘livereload’ do
watch(%r{preview_html/.+.(html)})
end

T> You might want to add *preview_html* to your .gitignore file as well.

Other Preview Apps

For Mac I recommend MouApp. For Linux there is
ReText

Remember to follow the golden rule of *Keeping it Simple*. Your Markdown productivity shouldn’t be tied to any
specific app. Choose things that complement your workflow not supplement it.

LaTeXing

LaTeXing

It’s no good being tied completely to Leanpub for publishing. Eventually we will want someway to locally generate
all the formats that Leanpub does. We have already covered generating HTML locally, but what about PDFs?

Kramdown doesn’t support PDFs natively, it uses an in-between format called LaTeX. So before we get started on
converting Markdown to PDFs we are going to have to learn some LaTeX. LaTeX knowledge is not strictly necessary
but it’s going to be beneficial to know how the magic works.

Magic is great and makes things easier, but when it fails, and you don’t know how it works, frustration finds its way
into your soul. Frustration kills the desire to create and will leave you a sad, bleeding, heaving mess on the floor.
Probably not that graphic, but it’s terrible, trust me.

Let’s avoid some future frustration and learn a little LaTeX.

Installing LaTeX

Before you dive in there are some things you need to know. The LaTeX community is a bit bloated. Well, maybe “a bit”
is an understatement, LaTeX makes Photoshop seem like an anorexic. This bloat is the result of the community being
really old in software years — TeX is older than I am. As a result of its

age, LaTeX has accumulated a ton of stuff.

LaTeX standard distributions are measured in the gigabytes! It has more GUIs than a Windows development community, more
packages than a Ruby community and an outdated package distribution model that makes Macports seem like homebrew. TeX
is what happens when an emacs-esque community spends its time writing technical papers and doesn't have VIM fights to
keep them busy.

Bloat in distributions alone wouldn't be so bad but the documentation is bloated as well. Want to add code
highlighting? There are a million different solutions and a million different suggestions. Want a GUI for previewing?
Pick one of dozens. Want an implementation that is more turing complete? Do you want your turing in LaTeX, Lua, or
Ruby?

I'm going to simplify things and tell you what to use. You don't have to ultimately follow my suggestions, but stick
with them until you figure out what you're doing. Fight the urge to over evaluate and pick the best solutions. Without
knowledge you won't be able to evaluate anything. Choosing from a place of ignorance is an exercise in futility. You'll
find that by diving in, you'll discover what you like — what your taste in tools is; you'll ultimately discover what
works for you.

On Mac

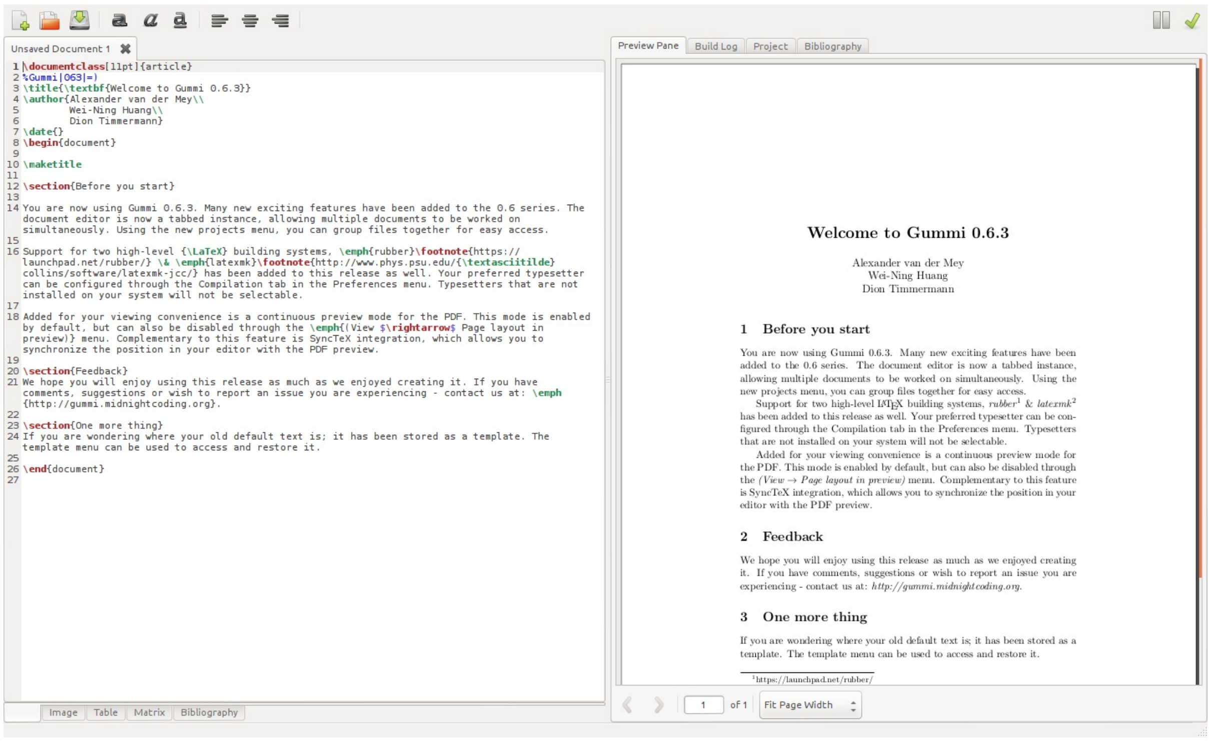
As much as I'd like to recommend you use a package manager like homebrew or macprots to keep TeX up to date, it's not a
good idea. Unfortunately, using homebrew would cause more problems with bugs and compatibility issues than you'd gain
from using it. The best way (and the official way) is to install LaTeX + TeX from mactex.org.
Just follow the instructions and you should be good to go.

On Ubuntu (and other Debians)

Installing LaTeX on Ubuntu is pretty straightforward and essentially involves running the following:

{:lang="sh"}
$ sudo apt-get install texlive

We will need a GUI for previewing the results. For this, we will use Gummi. Gummi is simpatico with the Markdown
philosophy of keeping previewing separate from editing. It includes separate tabs and panels for previewing, editing,
and examining the build log from `pdflatex` . It looks like this:



```latex
\documentclass[11pt]{article}
%Gummi|063|=)
\title{\textbf{Welcome to Gummi 0.6.3}}
\author{Alexander van der Mey\\
        Wei-Ning Huang\\
        Dion Timmermann}
\date{}
\begin{document}

\maketitle

\section{Before you start}

You are now using Gummi 0.6.3. Many new exciting features have been added to the 0.6 series. The document editor is now a tabbed instance, allowing multiple documents to be worked on simultaneously. Using the new projects menu, you can group files together for easy access.

Support for two high-level {\LaTeX} building systems, \emph{rubber}\footnote{https://launchpad.net/rubber/} \& \emph{latexmk}\footnote{http://www.phys.psu.edu/{\textasciitilde}collins/software/latexmk-jcc/} has been added to this release as well. Your preferred typesetter can be configured through the Compilation tab in the Preferences menu. Typesetters that are not installed on your system will not be selectable.

Added for your viewing convenience is a continuous preview mode for the PDF. This mode is enabled by default, but can also be disabled through the \emph{(View $\rightarrow$ Page layout in preview)} menu. Complementary to this feature is SyncTeX integration, which allows you to synchronize the position in your editor with the PDF preview.

\section{Feedback}
We hope you will enjoy using this release as much as we enjoyed creating it. If you have comments, suggestions or wish to report an issue you are experiencing - contact us at: \emph{http://gummi.midnightcoding.org}.

\section{One more thing}
If you are wondering where your old default text is; it has been stored as a template. The template menu can be used to access and restore it.

\end{document}
```

Welcome to Gummi 0.6.3

Alexander van der Mey
Wei-Ning Huang
Dion Timmermann

1 Before you start

You are now using Gummi 0.6.3. Many new exciting features have been added to the 0.6 series. The document editor is now a tabbed instance, allowing multiple documents to be worked on simultaneously. Using the new projects menu, you can group files together for easy access.

Support for two high-level LaTeX building systems, *rubber*[1] & *latexmk*[2] has been added to this release as well. Your preferred typesetter can be configured through the Compilation tab in the Preferences menu. Typesetters that are not installed on your system will not be selectable.

Added for your viewing convenience is a continuous preview mode for the PDF. This mode is enabled by default, but can also be disabled through the *(View → Page layout in preview)* menu. Complementary to this feature is SyncTeX integration, which allows you to synchronize the position in your editor with the PDF preview.

2 Feedback

We hope you will enjoy using this release as much as we enjoyed creating it. If you have comments, suggestions or wish to report an issue you are experiencing - contact us at: *http://gummi.midnightcoding.org*.

3 One more thing

If you are wondering where your old default text is; it has been stored as a template. The template menu can be used to access and restore it.

[1]https://launchpad.net/rubber/

{:center=""}

Get started by running the following:

{:lang="sh"}
$ sudo apt-get install gummi

APT will install the necessary prerequisites from the texlive package. Once installed, we can
proceed by getting up to speed with a test document. Open Gummi and you'll be presented with a test
document.

Installing LaTeX Packages on Mac

There is no management of LaTeX packages on Mac. To install them you

either use *MacTeXtras* from
mactex.org or manually copy them into your LaTeX path.

Installing LaTeX Packages on Ubuntu (and other Debians)

Ubuntu (and other Debian based distros) don't have tlmgr (the LaTeX package manager) and you will use *apt* to manage
LaTeX packages. You might be thinking: "This is ugly and stupid. I mean, Ruby has games, Lua has luarooks, Python has
peepee, Cocoa has Cocaine Pods, even PHP has something I cant remember the name of [^all-the-names]. Why cant LaTeX get
with the times?"

LaTeX's package manager is unfortunately more like apt than a LaTeX package manager and would cause conflicts if
utilized on Debian installs. If it was available it would do all sorts of screwy things to your system.

Not to worry though, there are gigantic repos [^facetious] of all the LaTeX packages you could ever want included in
apt. You can find them by searching for them:

{:lang="sh"}
$ apt-cache search texlive

To search for a specific package (in this example `listings`) do:

{:lang="sh"}
$ apt-cache search listings

Here are some things you'll probably want to install:

{:lang="sh"}
texlive-extra-utils - TeX Live: TeX auxiliary programs
texlive-font-utils - TeX Live: Graphics and font utilities
texlive-fonts-extra - TeX Live: Extra fonts
texlive-fonts-recommended - TeX Live: Recommended fonts

Install them with:

{:lang="sh"}
$ sudo apt-get install texlive-extra-utils

```
$ sudo apt-get install texlive-font-utils
$ sudo apt-get install texlive-fonts-extra
$ sudo apt-get install texlive-fonts-recommended
```

[^all-the-names]:
Gems for ruby, luarocks for lua, pypy for python, pear + composer for php, CocoaPods for cocoa respectively.

[^facetious]:
I'm being facetious. It's really not that big of a deal and works quite efficiently. It could be
better but it could also be way way worse. It's not like you will [see below] need to manage multiple versions of
LaTeX for different books. Unlike say a Ruby app, your final deliverable is more
like a creation of art than it is programming. Users don't care what version of LaTeX you used to generate your PDF
. Well, unless they're one of those PDF version snobs god I hate those guys. They even have their own hats and
handshakes now.

Using Packages

To use a package simply do the following:

{:lang="TeX"}
\usepackage{packagename}

For example:

{:lang="TeX"}
\usepackage{listings}

ctan.org is the place to lookup packages.

That is about all you need to know to get up and running. Have fun and break things.

LaTeX: A Very Brief Introduction

The first thing you need to know to grok LaTeX is that everything is done using back-slash tags — everything, yes,

everything. They look like this; `\title` , `\document` , `\newpage` etc. These *tags* are referred to as *commands*
remember that, it’s important for googling.

To pass an argument to a command we use braces e.g `\title{On the speed of Unladen Swallows in Vacuums}` . LaTeX
uses two different syntaxes for arguments, one for optional arguments and one required arguments. Content that will be operated on is passed via curly braces and the other arguments are passed via brackets e.g: `\textbf[Mono, 10pt]{BOld It}` . Each required argument ends up in it’s own curly brace like this: `\begin{arg}{arg}` .
Arguments in the curly braces are required and ones in brackets are optional.

Intuitively, you’re likely to think about LaTeX like you would a text markup language; tags being processed one
after another and the output being inserted in their place. You can avoid a lot of frustration if you think of LaTeX
more like a programming language than a markup language. Although LaTeX isn’t turing complete [^turing-completeness],
it can do a lot of crazy, very flexible, powerful, unexpected things.

Here is a quick example to give you a basic understanding of the programming nature of LaTeX:

{:lang=”TeX”}
\documentclass{article}

```
\title{Mein Dark Side: On The Struggle of Ruling the Galaxy and Baking Cookies}
\author{Darth Vader}
\date{\today{}}

\begin{document}
\maketitle

In a galaxy far far away. Okay, no cliche starts. I meant in a planet far far away. Or on a planet? Yeah, it was on a
planet. I'm sure of it. Or maybe some of the people were one the planet? And the others were like in it? Maybe, the
people in it (the planet) had gone crazy because of light deprivation? Like, they became cannibals and stuff.

\end{document}
```

The first command:

{:lang=”TeX”}
\documentclass{article}

Tells LaTeX to treat this as an *article* type.

Then we set some metadata:

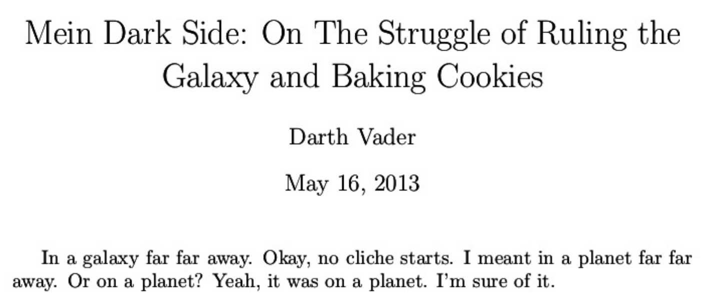
{:lang=”TeX”}
\title{Mein Dark Side: On The Struggle of Ruling the Galaxy and Baking Cookies}
\author{Darth Vader}
\date{\today{}}

This wont be displayed until we call the commands that output it. You can think of the above commands like variables.
A corresponding command has to be called later to output a variable. Much like a php *echo* or a ruby *puts*.

Open the document and call \maketitle to insert a properly formatted title:

{:lang=”TeX”}
\begin{document}
\maketitle
\end{document}

This results in the following output:



Mein Dark Side: On The Struggle of Ruling the Galaxy and Baking Cookies

Darth Vader

May 16, 2013

In a galaxy far far away. Okay, no cliche starts. I meant in a planet far far away. Or on a planet? Yeah, it was on a planet. I'm sure of it.

{:center=””}

A mixture of programming and markup language is a bit strange at first but very powerful once you get the hang of it. A
LaTeX document can be used to build articles, books, generate citations, hit lists (not recommended), html pages, table
of contents etc. There are commands and packages for accomplishing

practically any sort of output.

I’m not going to focus much on the commands available in LaTeX. Once you know the syntax you can learn the commands on
a need to know basis. Whenever you need a new command google it. Don’t understand a command? Look up its
documentation.

[^turing-completeness]:
I lied. Technically LaTeX is turing complete. However, in practice using LaTeX as a programming language is near
impossible, its syntax simply isn’t suited to it. There are however implementations that are programming friendly;
LuaTeX is the most popular one.

Custom LaTeX Styles

There are two types of things you can use to customize LaTeX. Style files, with the extension `.sty` and class files
with the extension `.cls` .

Style files, unlike their css brethren in the HTML world, do a lot more than handle styles. They’re basically a
collection of anything LaTeX. You can put macros, styles, commands etc in them and then reuse them as packages later.

They look a little like this:

{:lang=”TeX”}
\NeedsTeXFormat{LaTeX2e}[1994/06/01]
\ProvidesPackage{custom}[2013/05/13 Custom Package]

```
\newcommand{\cats}{}

\endinput
```

- \NeedsTeXFormat{…} Says I need this LaTeX version or I will destroy your TeX and make it look like barf.
- \ProvidesPackage Does what it says i.e details what the package is. The name (“custom”) must match the filename without the extension.
- \endinput You must close the package with this.

To use them you just call the `\usepackage` command:

{:lang="TeX"}
\usepackage{custom}

The package must be in your LaTeX path. Where your LaTeX path is, I have no idea. Probably
where you last left it? Retrace your steps? I digress.

Depending on your installation the LaTeX path is whatever `$ kpsewhich -var-value=TEXMFHOME` tells you.

What is *kpsewhich*? I'll let the docs tell you:

{:lang="sh"}
Standalone path lookup and expansion for Kpathsea.
The default is to look up each FILENAME in turn and report its
first match (if any) to standard output.

Ah, good old *Kpathsea*. I've been Kpathing since I was a wee little child. Actually no, but your sarcasm detector saw
that coming.

So, what is Kpathsea? Do I dare type `$ kpathsea` and risk falling further down the rabbit hole? I'm expecting a
definition like:

{:lang="sh"}
Standalone path tool and creator for Kpath

Let's brave the man anyway and see what we come up with:

{:lang="sh"}
$ kpathsea
zsh: command not found: kpathsea

Thank god it's something revealing. I feel like I'm learning the secrets of the Kpathsea universe. What wonders abound
I wonder? Will I discover the secret to immortality? The properties of God? The formula for a new face enrichment
cream?

A quick google reveals the following:

```
## Kpathsea

Kpathsea is a library to do path searching. It is used in the Web2C implementation of TeX and friends.
```

[^cream]

Aha! Kpathsea does exactly what path and your shell accomplish. Which actually makes sense, because TeX came before
modern shells. TeX was first and had to invent its own stuff. Imagine how different documentation would have been had
Kpathsea become a standard. Just a reminder to append rbenv shims to your Kpathsea. Okay, I digress.

We wont use packages for our purposes, but it's a good idea that we now know the basics of how they
work. Now about those classes.

You know that line we see at the beginning of every LaTeX document? The one that looks like:

{:lang="TeX"}
\documentclass{scrartcl}

This tells the document to use that class. A class can be an; article, report, paper, book etc. A book class might have
macros for Chapter and Covers. An article class might have commands for references, indexes, footnotes etc.

A class looks like this:

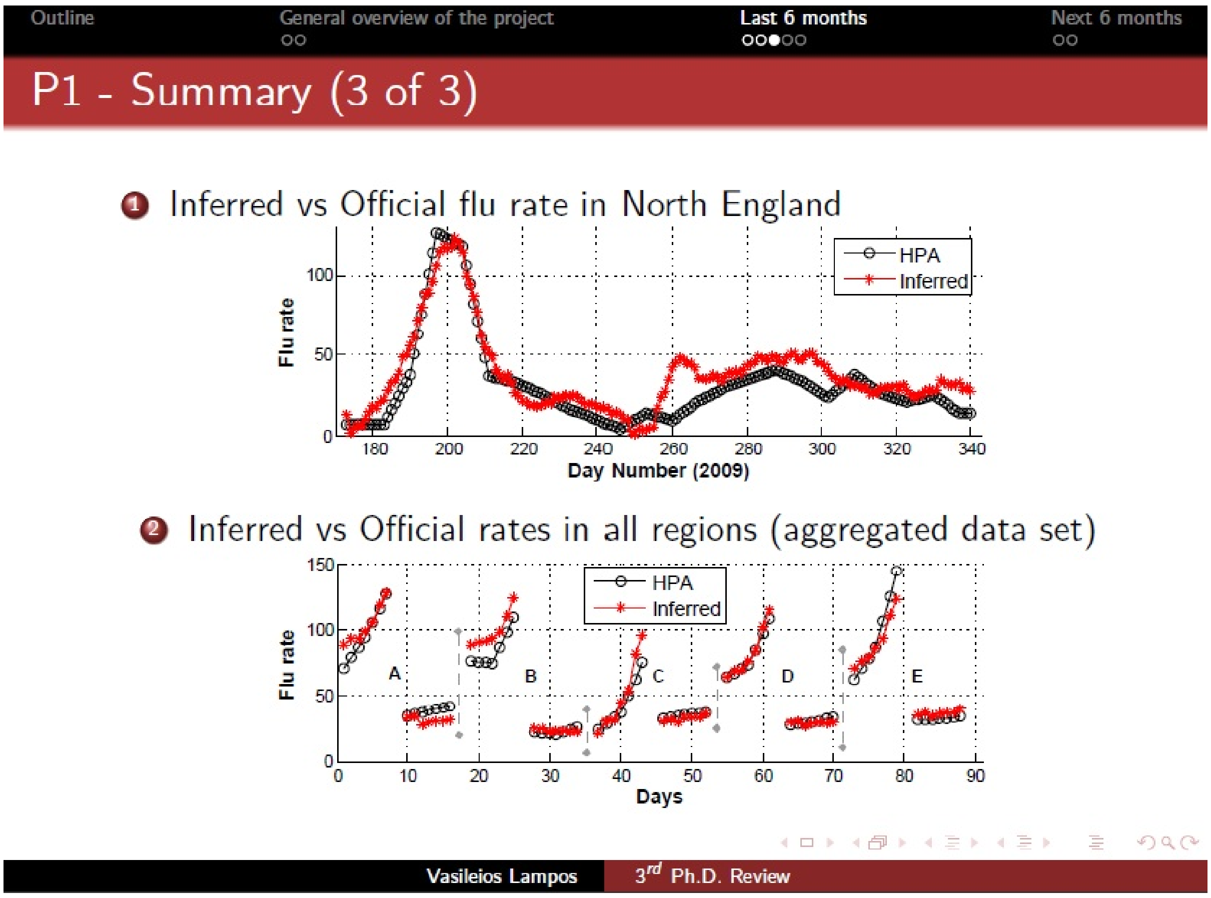
{:lang="TeX"}
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{myclass}[2013/5/5 My Class]

```
%% Load base class
\LoadClass[a4paper]{article}

\endinput
```

Pretty much the same as a package but with a slightly different definition syntax.

One can completely change the look and appearance of a document just by changing its class. With classes alone you can
create everything from slides which look awesome, to books, which look like what you're reading now.

Yes, I said slides. They look like:



P1 - Summary (3 of 3)

1. Inferred vs Official flu rate in North England
2. Inferred vs Official rates in all regions (aggregated data set)

{:center=""}

They are created with code like:

{:lang="TeX"}
\documentclass{beamer}

```
\begin{document}
  \begin{frame}
    \frametitle{This is the first slide}
    %Content goes here
  \end{frame}
  \begin{frame}
    \frametitle{This is the second slide}
    \framesubtitle{A bit more information about this}
    %More content goes here
  \end{frame}
% etc
\end{document}
```

Classes open up a whole range of options for completely customizing the output of a document. The possibilities are
beyond the scope of this book but I encourage you to go forth into the

internets and learn.

[^cream]:
Markdown added for effect. The original was HTML which was quite boring
and verbose. Grow up Chrome, learn
a real markup language like Markdown. Quick thought to ponder: What if the
web was a simple text based language instead of XML? Would we have had
near the level of innovation? Would flash still dominate site creation?

Going Local

Going Local

Generating the formats Leanpub does is fairly straightforward [^pdf-exceptions]. Kramdown supports pretty much all
formats via its LaTeX and HTML converters. What is going to be a bit more involved is replicating all the styling
options and workflow Leanpub supports. For that we are going to need to get our hands dirty with a little ruby.

T> You could if you like skip this chapter and simply download my krambook gem.
T> I'd recommend you at least skim over the chapter though so you get a basic idea of how to customize a markdown
T> parser. If you go through all the effort of writing an ebook chances are you are going to want to do at least
T> some custom stuff.

All the Markdown in One

First things first, we need a script that takes our Book.txt, parses the index and then joins all the Markdown files
into one. We can then pass the joined file into Kramdown to get LaTeX back. Why LaTeX? Kramdown doesn't support support
PDF, which means our starting format for PDF generation is always going to be LaTeX.

We will develop the script to be used from IRB first, then we will wrap it all up as command line script and finally,
as a gem.

The first thing we need to do is read Book.txt as a string, then split the lines and white space; giving us an array of
files to load and parse.

First some vars to hold things:

```
{:lang=”ruby”}
files = []
joined_file_path = File.expand_path ‘./manuscript/‘ + ‘Joined.md’
```

Now we need to parse our index file (Book.txt) and get the names of the files:

```
{:lang=”ruby”}
File.open(File.expand_path(‘./manuscript/Book.txt’), ‘r’) do |f|
f.each_line do |line|
files << line.gsub(/\s+/, “”)
end
end
```

This opens the file for reading, loops through each line, strips the line of whitespace and then adds it to a *files*
array.

Now we join all the files and output them into a single Markdown file:

```
{:lang=”ruby”}
File.open(joined_file_path, ‘a’) do |f|
files.each do |file_path|
full_file_path = File.expand_path ‘/manuscript/‘ + file_path
f.puts ‘’
f.puts IO.read(full_file_path) if File.exists?(full_file_path)
end
end
```

This opens an output file in append mode, loops through our files array, reads each file into a new string and appends
the content to the output file.

There is one problem though, we don’t check for an existing `Joined.md` file. Let’s add a check for that now:

```
{:lang=”ruby”}
File.delete(joined_file_path) if File.exist?(joined_file_path)
```

The full ruby script now looks like:

```
{:lang=”ruby”}
files = []
```

joined_file_path = File.expand_path ‘./manuscript/‘ + ‘Joined.md’
File.delete(joined_file_path) if File.exist?(joined_file_path)

```ruby
File.open(File.expand_path('./manuscript/Book.txt'), 'r') do |f|
  f.each_line do |line|
    files << line.gsub(/\s+/, "")
  end
end

File.open(joined_file_path, 'a') do |f|
  files.each do |file_path|
    full_file_path = File.expand_path './manuscript/' + file_path
    f.puts ''
    f.puts IO.read(full_file_path) if File.exists?(full_file_path)
  end
end
```

Save the file as `join_manuscript_files.rb` to a folder. Copy over some test content from a Sample Leanpub project and
then boot up irb:

{:lang=”sh”}
$ irb

We can run the script by simply requiring it:

{:lang=”ruby”}
require ‘kramdown’
require ‘./join_manuscript_files.rb’

If you open *Joined.md* you’ll notice one problem, on the very top of the file there will be an extra new line
. We can fix the this by adding a check for the last line to our files loop. T

he files loop now looks like:

{:lang=”ruby”}
files.each do |file_path|
full_file_path = File.expand_path ‘./manuscript/‘ + file_path
f.puts IO.read(full_file_path) if File.exists?(full_file_path)
f.puts ‘’ unless file_path == files.last
end

There is a further use case for this joining magic. We can utilize it to

convert to another Markdown flavor before
processing. For example, we might be using a slightly different flavor of
Markdown and want to pre-process it into
Kramdown first. Instead of just loading the file and joining the strings,
we can first parse it though a Kramdown
converter:

```
{:lang=”ruby”}
f.puts Kramdown::Document.new(IO.read(full_file_path)).to_kramdown if
File.exists?(full_file_path)
```

The script would then look like:

```
{:lang=”ruby”}
files.each do |file_path|
full_file_path = File.expand_path ‘./manuscript/‘ + file_path
f.puts Kramdown::Document.new(IO.read(full_file_path)).to_kramdown if
File.exists?(full_file_path)
f.puts ‘’ unless file_path == files.last
end
```

We can easily swap out formats by changing the `to_format` call e.g the
following would output a joined file of LaTeX:

```
{:lang=”ruby”}
f.puts Kramdown::Document.new(IO.read(full_file_path)).to_latex if
File.exists?(full_file_path)
```

The does it for now. Later we will wrap this up by creating a gem for the
script.

Creating PDFs

Once we have a joined Markdown file we can turn it into LaTeX and then
into a PDF. To turn it into LaTeX we can simply
pass it (the joined file) into the Kramdown command line tool. The
Kramdown bin takes two arguments:

```
{:lang=”sh”}
-i, —input ARG
Specify the input format: kramdown (default) or html
-o, —output ARG
Specify one or more output formats separated by commas: html (default),
```

kramdown, latex or remove_html_tags

T> The Kramdown CLI takes arguments using a single dash and arguments passed to them are designated using two dashes
T> e.g `-o latex --template document`

Kramdown will output to the stdout (the terminal) so we will need to pipe it to a file:

{:lang="sh"}
$ kramdown Joined.md -o latex > Joined.tex

Now we just need to utilize `pdflatex` to convert LaTeX to PDF. The command for this is simple:

{:lang="sh"}
$ pdflatex Joined.tex

After running this we will get all sorts of errors:

{:lang="sh"}
! LaTeX Error: Missing \begin{document}.
…more errors

Open up *Joined.tex* and you will discover that the LaTeX generated by Kramdown is incomplete; it lacks inclusion
statements for packages, it lacks a begin and end document etc. How do we fix this? Well, it turns out that *kramdown*
uses templates to get wrapping around documents:

{:lang="sh"}
—template ARG
The name of an ERB template file that should be used to wrap the output

```
This is used to wrap the output in an environment so that the output can
be used as a stand-alone document. For example, an HTML template would
provide the needed header and body tags so that the whole output is a
valid HTML file. If no template is specified, the output will be just
the converted text.

When resolving the template file, the given template name is used first.
If such a file is not found, the converter extension is appended. If the
file still cannot be found, the templates name is interpreted as a
template name that is provided by kramdown (without the converter
extension).

```

```
kramdown provides a default template named 'document' for each converter.

Default: ''
Used by: all converters
```

By default, when you convert to LaTeX or html you get fragments. This is useful if you want to add your own wrapping,
e.g when generating an html book or parsing Markdown for blog posts. Personally, I think it would be a good idea to
default to the document template when using Kramdown via cli, but I digress.

To generate a proper LaTeX file we just have to do:

{:lang="sh"}
$ kramdown Joined.md –template document -o latex > Joined.tex

Opening *Joined.tex* now reveals a properly formatted file.

Let's run pdflatex again:

{:lang="sh"}
$ pdflatex Joined.tex

All appears to have ran correctly!

Now open the generated pdf:

{:lang="sh"}
$ xdg-open Joined.pdf

facepalm There is a noticeable lack of a table of contents. Now before you consider taking up ebook crafting with a
typesetter and typewriter, let's do some googling, well I'll do the googling, you can just read onward.

It turns out that pdflatex needs to be ran more than once to generate a proper document. It needs to go through once
to find all the references, page numbers and footnotes etc, then a second time to add them.

A> pdflatex stores its state and logs in a variety of files in the same directory you run
A> it in. You might consider using a separate output directory, the manuscript folder can get a bit

A> messy otherwise.

Images and Your PDFs.

Let’s see what happens when we add some images. We will include them in our Markdown and then go through the conversion process again.

Add the following (make sure the image is relative to the directory the file is in) to your Markdown:

{:lang=”md”}

Now run:

{:lang=”sh”}
$ kramdown Joined.md —template document -o latex > Joined.tex

Then:

{:lang=”sh”}
$ pdflatex Joined.tex

Open it up:

{:lang=”sh”}
$ xdg-open Joined.pdf

We get the following output:

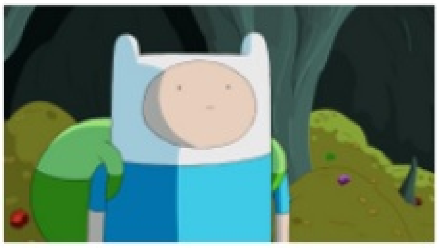


Figure 1: Adventure Time

{:center=””}

This is pretty good but there are going to be some cases when we need the

image inline, not re-inserted at the top
of the document. First things first, let’s figure out what caused this in the TeX file:

{:lang=”TeX”}
Briefly, to write a Leanpub book, you write a bunch of plain text files…..

```
\begin{figure}
\begin{center}
\includegraphics{http://static.bookstack.cn/projects/markdown-to-
ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-
ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-
ebook/manuscript/images/Finn.png}
\end{center}
\caption{Adventure Time}
\end{figure}

To learn more about how to write a Leanpub book....
```

At least to my untrained eyes, it looks like the `\begin{figure}` and `\end{figure}` is the source of the problem.
Let’s manually remove this and regenerate the file to see what happens:

The TeX should now look like:

{:lang=”TeX”}
\begin{center}
\includegraphics{http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/images/Finn.png}
\end{center}

I> I removed the caption which was part of figure section.

If we run the conversion again then we get the desired output. Obviously, we cant manually do this every time we add an
image, so how do we construct our Markdown so it happens automatically?
Let’s open up `kramdown/converter/latex.rb`
and search for `figure` .

The source for the `convert_standalone_image` method reveals what is up:

{:lang=”ruby”}

```
# Helper method used by +convert_p+ to convert a paragraph that only contains a single :img
# element.
def convert_standalone_image(el, opts, img)
  attrs = attribute_list(el)
  "\\begin{figure}#{attrs}\n\\begin{center}\n#{img}\n\\end{center}\n\\caption{#
{escape(el.children.first.attr['alt'])}}\n#{latex_link_target(el, true)}\n\\end{figure}#
{attrs}\n"
end
```

The key is the comment:

```
Helper method used by +convert_p+ to convert a paragraph that only contains a single :img
element.
```

We need to place this image in a paragraph like so:

{:lang="md"}
Briefly, to write a Leanpub book, you write a bunch of plain text files, all in the same folder as this file. These text files are written in a format called Markdown, which is a simple way of doing formatting that

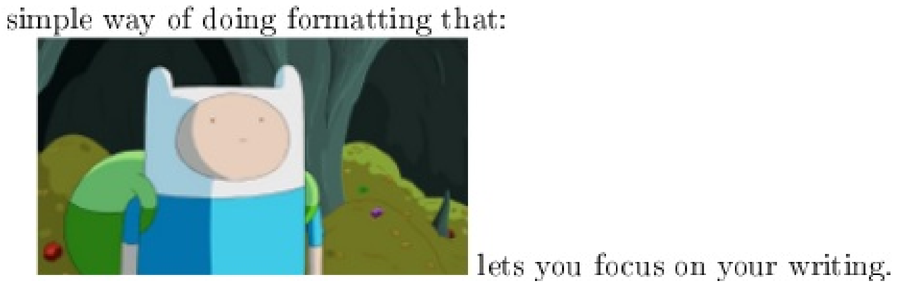
lets you focus on your writing.

VS (the figure way):

{:lang="md"}
Briefly, to write a Leanpub book, you write a bunch of plain text files, all in the same folder as this file. These text files are written in a format called Markdown, which is a simple way of doing formatting that lets you focus on your writing.

```
![Adventure Time](http://static.bookstack.cn/projects/markdown-to-
ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-
ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-
ebook/manuscript/images/Finn.png)
```

After regenerating we get:


simple way of doing formatting that:

lets you focus on your writing.

{:center=""}

There are still some problems. First of all, we would like this centered. How can we do that?

Looking at *Joined.tex* reveals we have lost our center command:

{:lang="TeX"}
\includegraphics{http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/images/Finn.png} lets..

Adding it back in like so:

{:lang="TeX"}
\begin{center}
\includegraphics{http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/http://static.bookstack.cn/projects/markdown-to-ebook/manuscript/images/Finn.png}
\end{center}

and then regenerating, results in:

{:center=""}

We are going to have to rectify this by patching the LaTeX converter to center things automatically. The syntax for
this will look like:

{:lang="md"}

{:center=""}

We could simply monkeypatch this onto Kramdown but we want the feature available via cli and to achieve that requires
modifying Kramdown directly. Fork it on github and then change *convert_img* (in kramdown/converter/latex.rb) to:

{:lang="ruby"}
def convert_img(el, opts)
if el.attr['src'] =~ /^(https?|ftps?):\/\//
warning("Cannot include non-local image")
''
elsif !el.attr['src'].empty?

```
@data[:packages] << ‘graphicx’
if el.attr[‘center’]
“#{latex_link_target(el)}\begin{center}\includegraphics{#
{el.attr[‘src’]}}\end{center}”
else
“#{latex_link_target(el)}\includegraphics{#{el.attr[‘src’]}}”
end
else
warning(“Cannot include image with empty path”)
‘’
end
end
```

Commit and push the changes, then run:

```
{:lang=”sh”}
$ rake install
```

```
A> This will install Kramdown from the repo.
A>
A> If you are running rbenv make sure you run:
A>
A> {:lang=”sh”}
A> $ rbenv rehash
A>
A> after installing.
```

Run the whole generation process again:

```
{:lang=”sh”}
$ kramdown Joined.md —template document -o latex > Joined.tex
```

Then:

```
{:lang=”sh”}
$ pdflatex Joined.tex
```

Open it up:

```
{:lang=”sh”}
$ xdg-open Joined.pdf
```

And we get:

{:center=””}

```
[^pdf-exceptions]:
With the exception of PDF most the ebook formats are rather simple. Most
if not all of them are at
their core just wrappers around html. Even the kindle format is just mobi
with Amazon’s crap wrapped around it.
Once we have HTML + PDF it’s pretty damn easy to support everything from
that starting point; even if we have to
code something ourselves, it will be minimal.
```

Common Customizations

Common Customizations

Chances are you are going to need to customize some of your output at some point. Thankfully, Kramdown is easily
extended to add new features. Any customization we make will involve modifying Kramdown in some fashion. We will
customize the converter not the converted. Customizing things once they’re already converted is unmaintainable and
would destroy all the delight in a work flow.

I> This chapter could be skipped if all you want to do is accomplish the customizations and not learn how they’re done
I> All the customizations I made are in my fork of kramdown in the
I> markdown-to-ebook branch just clone and follow the README to install.

My God, it’s full of Snippets

Something completely expected and yet completely unfortunate happens when you write a book on programming, you end up
with a ton of code blocks. Like a redhead from beyond the wall they become difficult to manage, yet, you are somehow
drawn to them over and over.

Just when you think you have everything under control your users demand full length source files. “Copying and pasting
is for those with free time!” “Give us source files!” Your readers shout! What is an author to do?

Any writer worth his salt preempts this reader rebellion and creates two separate things. He places code examples in
external files and in code blocks. He delicately manages them, keeping them in sync all the time. Inevitably he'll slip
up, one of his code blocks will contain some random gibberish where he typed `def igiveupkillmenowcatssalkajddewoq(published='wat')` and then forgot to change it.

A slip-up like this can ruin careers. The author ends up homeless, wearing only Abercrombie & Fitch and coding a whore
management system for a pimp in the shadiest part of town. He develops a cocaine addiction and ends up dead in a pool
of his own sweat.

Programmer politicians use his death to advocate for better work circumstances for developers. His tragic story is
covered in major magazines across the country. Cold, heartless, comments are made on Reddit and HN, they say that if
he had only used LaTeX none of this would have happened.

User alevkrf2 writes:

```
LaTeX has included this functionality for almost a decade. You can just do `\texttt{src/hello.c}` and LaTeX will
include the file no problem. With a custom macro it is even easier. Something like `\filename{src/hello.c}` isn't
hard at all to create.
```

User throwaway234 replies:

```
LaTeX is not a solution period. If you are going to provide LaTeX based solutions you might as well  not be providing a solution at all.
```

User catsanddogs replies:

```
@throwaway234 I personally see nothing wrong with LaTeX. The right tool for the right job. What is your problem
with LaTeX?
```

User throwaway234 replies:

```
Oh I don't know, how about that the technology is 2 decades old, comes with confusing documentation,  there are
like 50 zillion implantations, it throws random tags through my text etc. It's a bastardized language that  tries
```

```
to be something between turing-complete and plain txt.
```

User throwaway234 writes:

```
*implementations
```

User kawpo246 replies:

```
The things you listed as criticisms are some of the exact reasons others choose LaTeX. old = tried and battle
tested. confusing docs = detailed docs for technical users.

when you're using this to publish research papers that can change governments and entire economies, something old
and thoroughly documented is important. Blank is a prime example of this. A half-turing lang is really a matter of
taste and I don't see how it's a relevant criticism.
```

User noddywood writes:

```
I see these deaths as a worrying trend. How many more deaths do we need before the problem is addressed?
```

User rubynode.clojure replies:

```
@noddywood You are committing a blank blank fallacy. These stats seem large because they get so much    attention,
but I just don't see it as a real problem. There is no real evidence it is a real problem. A few outlier   deaths
from inappropriate Markdown usage and the ensuing backlash does not make an epidemic.
```

User dnret writes:

```
I think this is inevitable. He should have chosen to write Ruby instead.
He deserved it.
```

User dnret writes:

```
Why is everyone down voting me?
```

Including Source Files With Kramdown

Okay, maybe it wouldn’t have been that bad, but you should really learn

how to include external files in case something
terrible happens. Out of the box no Markdown processors support this but
it is scarily easy to add to Kramdown. Let’s
take a look at how.

First, we are going to need to decide on a syntax. We are going to need
two things;

1. A command telling our parser to include a file.
2. A file attribute that contains the file to be included.

Conveniently, there is a standard implemented by Kramdown already, that
defines custom attributes for an element. It
looks like this:

{:lang=”md”}
block
{:title=”The blockquote title”}

Our syntax will thus look like this:

{:lang=”md”}

```
~~~ ruby
~~~
{:include="external_file.rb"}
```

You might want to rush ahead and just implement a custom renderer for
Kramdown but we need to consider some things
first. For Leanpub to process things correctly we will need to have our
codeblocks possess the actual code in them. If
the Markdown files don’t have the actual code in their blocks, then when
they’re sent to Leanpub they cant be
processed. The only way to make Leanpub happy is pre-process our files and
insert the code in the blocks.

This means we need to write two renderers; one that churns out modified
Markdown with the code inserted into the
codeblocks and another that simply overrides the LaTeX renderer. This will
give us two flexible workflows; one for
Leanpub and one entirely local.

A Markdown Kramdown Renderer

The first thing we need to do is figure out how a Kramdown renderer works. It might seem complicated, but it’s all
pretty straightforward. API wise most (including Kramdown) of the Ruby Markdown renderers borrow from
Maruku.

In Maruku the rendering comes down to two tasks;

1. A processor (parser) that turns the Markdown into elements — into an AST.
2. Then the rendering class (in Kramdown its called a Converter) is called for each one of these elements.

Since we already have a codeblock element it’s going to be rather easy to grab its attributes, include the file and
return it. We wont have to bother with the parsing bits at all.

The best way to figure out what we need to modify is to look at some of the actual code for converting codeblocks. We
can open kramdown/converter/html.rb and take a look. A quick search in the file for *_codeblock* turns up the
following:

```
{:lang=”ruby”}
def convert_codeblock(el, indent)
attr = el.attr.dup
lang = extract_code_language!(attr)
if @coderay_enabled && (lang || @options[:coderay_default_lang])
opts = {:wrap => @options[:coderay_wrap], :line_numbers =>
@options[:coderay_line_numbers],
:line_number_start => @options[:coderay_line_number_start], :tab_width =>
@options[:coderay_tab_width],
:bold_every => @options[:coderay_bold_every], :css =>
@options[:coderay_css]}
lang = (lang || @options[:coderay_default_lang]).to_sym
result = CodeRay.scan(el.value, lang).html(opts).chomp << “\n”
“#{‘ ‘indent}#{result}#{‘ ‘indent}\n”
else
result = escape_html(el.value)
result.chomp!
if el.attr[‘class’].to_s =~ /\bshow-whitespaces\b/
result.gsub!(/(?:(^[ \t]+)|([ \t]+$)|([ \t]+))/) do |m|
suffix = ($1 ? ‘-l’ : ($2 ? ‘-r’ : ‘’))
```

```
m.scan(/./).map do |c|
case c
when “\t” then “\t“
when “ “ then “·“
end
end.join(‘’)
end
end
code_attr = {}
code_attr[‘class’] = “language-#{lang}” if lang
“#{‘ ‘*indent}#{result}\n\n”
end
end
```

This grabs the lang as a symbol, processes it with coderay and returns the HTML. To simplify our job a bit, we are
going to extend this method rather than recode it. We will parse the attributes for the inclusion, pull the relevant
file and insert onto the element’s content, then we can just call super and let it handle the rest.

First things first, we need to create a class that extends `Kramdown:□:Kramdown` :

```
{:lang=”ruby”}
module Kramdown
module Converter

class Includey < :□□:Kramdown
def convert_codeblock(el, indent)
return super(el, indent)
end
end
end
end
```

Now where do we get our include attribute from? My first guess is that we will find it on `el.attr` . We are going
to create some test files and play with them until we figure it out. Once we figure out how to get our attribute
everything else is going to be easy.

Create a new folder for our experimentation to live in and then enter it yourself. Don’t live it in it though, there is
nasty fellow named Clue that looks like Jeff Bridges that will attempt to wipe you out. No digitizer necessary, a
simple `$ cd` will suffice:

```
{:lang=”sh”}
$ mkdir includey_experiment
$ cd includey_experiment
```

Create a markdown file called `test.md` with the following:

{:lang=”md”}

~~~
```ruby
```
{:include="external_file.rb"}
~~~

Add a file called `exteneral_file.rb` with some sample ruby:

```
{:lang=”ruby”}
cats = ‘dog haters’
penguins = ‘ice lovers’
birds = ‘fluffy soft dinosaurs’
```

Then create a file for our converter and call it *test.rb*:

```
{:lang=”ruby”}
require ‘kramdown’
module Kramdown
module Converter

class Includey < :□□:Html
def convert_codeblock(el, indent)
puts el.attr.inspect
return super(el, indent)
end
end
end
end
```

Boot up irb in this directory:

{:lang=”sh”}

```
$ irb
```

To use our test converter we just need to require *test.rb* and then call our converter class.

{:lang="ruby"}

```
require './test.rb'
Kramdown::Document.new(IO.read('./test.md')).to_includey
```

ethod_missing on *Kramdown::Document* handles instantiating our converter. We should now (hopefully) see some output
in irb:

{:lang="sh"}

… Unfortunately, we get nothing. The first thing we need to do is make sure we are using the correct converter
class by `puts` ing the name of whatever is passed to *method_missing*:

{:lang="ruby"}

```
module Kramdown
class Document
def methodmissing(id, *attr, &block)
if id.to_s =~ /^to(\w+)$/ && (name = Utils.camelize($1)) &&
Converter.const_defined?(name)
puts name.inspect
output, warnings = Converter.const_get(name).convert(@root, @options)
@warnings.concat(warnings)
output
else
super
end
end
end
end
```

We should see *Includey* outputted somewhere.

Now we have to figure out if we are hitting any methods in our converter class. Let's add some more test text to our
Markdown file:

{:lang="md"}

```
# Heading
```

Paragraph

# Code sample

```ruby
nemo = 'missing'
```

```ruby
```
{:include="external_file.rb"}
```

Override some more methods like this:

{:lang="ruby"}
def convert_text(el, indent)
puts ‘in text’
escape_html(el.value, :text)
end

```
def convert_p(el, indent)
  puts 'in p'
  if el.options[:transparent]
    inner(el, indent)
  else
    format_as_block_html(el.type, el.attr, inner(el, indent), indent)
  end
end
```

Run the following again in irb:

{:lang="ruby"}
require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey

And we get back:

{:lang="sh"}
in text
in p
in text
in text
in p
in p
```

So we are a overriding, we are just not hitting the correct method. Let’s try overriding the convert method to see what
elements we are hitting:

{:lang=”ruby”}

```
# Dispatch the conversion of the element +el+ to a +convert_TYPE+ method using the +type+ of
# the element.
def convert(el, indent = -@indent)
  puts el.type
  send(DISPATCHER[el.type], el, indent)
end
```

Run this again in irb:

{:lang=”ruby”}
require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey

We only get back:

{:lang=”sh”}
root

Well, that is not very helpful. Let’s take a shot into the dark and try overriding the codespan method instead:

{:lang=”ruby”}
def convert_codespan(el, indent)
puts ‘in codespan’
super(el, indent)
end

Run it again in irb:

{:lang=”ruby”}
require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey

{:lang=”sh”}
…
in codespan
…

And bingo! It seems we overwrote the wrong thing. Duh! Rewrite our *test.rb* as the following:

```
{:lang=”ruby”}
require ‘kramdown’
module Kramdown
module Converter

class Includey < :□□:Html
def convert_codespan(el, indent)
puts ‘here’
puts el.attr.inspect
return super(el, indent)
end
end
end
end
```

Then try again:

```
{:lang=”ruby”}
require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey
```

Now we get back:

```
{:lang=”sh”}
here

{}
here


{}
```

This is interesting, it seems that we don’t have any attributes available yet. Let’s take a look at the original *convert_codespan* and see where it extracts the language. The key line is:

```
{:lang=”ruby”}
lang = extract_code_language(el.attr)
```

If we take a look at *extract_code_language* we find:

```
attr['class']
```

The usage of brackets for accessing elements indicates that *attr* is a hash or at least hash like. This makes me
wonder what we have going on in the *attr*. We can probably get an idea if we find the relevant class. Let’s inspect
attr for its class by doing the following:

{:lang=”ruby”}
def convert_codespan(el, indent)
puts ‘in codespan’
puts el.attr.class
return super(el, indent)
end

Run things in irb again:

{:lang=”ruby”}
require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey

We get back:

{:lang=”sh”}
in codespan
Hash

It seems we have a Hash on our hands. Which means, at some point this element is parsed and has its attributes set to
the hash. We need to find that bit of code and see how it works.

If we go into the *Element* class we will find the following lines:

{:lang=”ruby”}

```
# The attributes of the element. Uses an Utils::OrderedHash to retain the insertion order.
def attr
  @attr ||= Utils::OrderedHash.new
end
```

We need to figure out the point where IAL’s (Inline Attribute Lists) get turned into attributes. A search through the
code for *IAL* turns up:

{:lang=”ruby”}
IAL_BLOCK = /{:(?!:|\/)(#{ALD_ANY_CHARS}+)}\s*?\n/
IAL_BLOCK_START = /^#{OPT_SPACE}#{IAL_BLOCK}/

Also a method that parses them:

{:lang="ruby"}
def parse_block_extensions
if @src.scan(ALD_START)
parse_attribute_list(@src[2], @alds[@src[1]] ||= Utils::OrderedHash.new)
@tree.children << Element.new(:eob, :ald)
true
end
end

The interesting part is the mention of *parse_attribute_list*. Let’s inspect that out to stdout:

{:lang="sh"}

Well now, there is not much we can do with this, it’s ummmm… blank. Where do we go from here? Let’s modify the
attributes string in our markdown to look like this:

{:lang="md"}

~~~
# Heading

Paragraph

# Code sample

```ruby
nemo = 'missing'
```
{: .language-ruby}
~~~

Let’s run it again via irb and make sure we are getting class attributes out to our html:

{:lang="html"}

Heading

\n\n
Paragraph

\n\n

Code sample

\n\n
`ruby\nnemo = ‘missing’\n`

\n

We are, so let’s go back to *extract_code_language* and add the following inspects for debugging:

```
{:lang=”ruby”}
require ‘kramdown’
module Kramdown
module Converter

class Includey < :□□:Html
def convert_codespan(el, indent)
puts ‘here’
puts el.inspect
puts el.attr.inspect
lang = extract_code_language(el.attr)
puts lang.inspect
return super(el, indent)
end
end
end
end
```

Run it in irb again:

```
{:lang=”ruby”}
require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey
```

We get:

```
{:lang=”sh”}
here

{}
nil
=> “
```

Heading

\n\n
Paragraph

\n\n

Code sample

\n\n
`ruby\nnemo = ‘missing’\n`

\n”

This indicates our attributes have not yet been parsed. Yet somehow, they end up in our output. It must be happening in
the last line! Let’s take a look at *format_as_span_html*:

{:lang=”ruby”}

```ruby
# Format the given element as span HTML.
def format_as_span_html(name, attr, body)
  "<#{name}#{html_attributes(attr)}>#{body}</#{name}>"
end
```

Bazinga! Seems `html_attributes` is what we are looking for. Let’s take a look at it:

{:lang=”ruby”}

```ruby
# Return the HTML representation of the attributes +attr+.
def html_attributes(attr)
  attr.map {|k,v| v.nil? || (k == 'id' && v.strip.empty?) ? '' : " #{k}=\"#{escape_html(v.to_s,
:attribute)}\"" }.join('')
end
```

WAT! It just does a map against the attr — how in the world do we end up with the class on our element then? I mean
it’s right there just look:

{:lang=”html”}

`ruby\nnemo = ‘missing’\n`

\n

It’s at this moment when you realize what you had missed all along. Everything comes back around and everything makes
sense. Like at the end of a great crime thriller you can see all the connections. You shout: “it was the guy with the
gimp all along! He was in the crowd with the civilians! The killer was the, omg I knew it, but I didn’t, I’m
surprised, but I’m not. I could feel it coming.”

It’s the best kind of feeling to finally *see* but it’s the worst kind of feeling because you know you missed so much.

What did I miss? Way back when I started this I was using the wrong syntax. We got codespans not codeblocks because I
was using the wrong syntax. The dead ringer should have been the output of the *ruby* language code into the block.

We are back on the right track though. We can see clearly now.

Let’s modify the Markdown to look like this:

{:lang=”md”}

```
# Heading

Paragraph

# Code sample

~~~ ruby
nemo = 'missing'
~~~
{:include='external_file.rb'}
```

And our ruby like this:

{:lang=”ruby”}
require ‘kramdown’
module Kramdown
module Converter

class Includey < :□□:Html
def convert_codeblock(el, indent)
puts ‘in codeblock’

```
puts el.attr.inspect
return super(el, indent)
end
end
end
end
```

Run in irb again:

```
{:lang=”ruby”}
require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey
```

And we get:

```
{:lang=”sh”}
in codeblock
{“class”=>”language-ruby”, “include”=>”external_file.rb”}
```

We are now getting our attributes. What we have to do now is modify the content with the external file. Let’s do the
following:

```
{:lang=”ruby”}
require ‘kramdown’
module Kramdown
module Converter

class Includey < :□□:Html
def convert_codeblock(el, indent)
if el.attr.include?(‘include’)
file_path = File.expand_path(Dir.pwd + ‘/‘ + el.attr[‘include’])
el.value = IO.read(file_path) if File.exists?(file_path)
end
```

```
        return super(el, indent)
      end
    end
  end
end
```

Run it in irb again:

```
{:lang=”ruby”}
```

require ‘./test.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey

We get back the following HTML:

{:lang=”html”}

Heading

\n\n
Paragraph

\n\n

Code sample

\n\n
\n

```
1cats     =
    ‘dog haters‘\n2penguins = ‘ice
    lovers‘\n3birds    = ‘fluffy
    soft dinosaurs‘\n
```

\n
\n
\n

Bingo! Now the next step is to build a parser that renderers Markdown files with the code included. This should be
really straightforward. All we need to do is a return a Markdown file from Kramdown which we can then re-save to an
output directory. To do this we need to first inherit from the `Kramdown` converter:

{:lang=”ruby”}
require ‘kramdown’
module Kramdown
module Converter

class Includey < : :Kramdown
def convert_codeblock(el, indent)

if el.attr.include?(‘include’)
file_path = File.expand_path(Dir.pwd + ‘/‘ + el.attr[‘include’])
el.value = IO.read(file_path) if File.exists?(file_path)
end

```
          return super(el, indent)
        end
      end
    end
  end
```

Save as `test_md_render.rb` and then run with:

{:lang=”ruby”}
require ‘./test_md_render.rb’
Kramdown::Document.new(IO.read(‘./test.md’)).to_includey

No we get back:

{:lang=”md”}

```
# Heading\n\nParagraph\n\n# Code sample\n\n    cats     = 'dog haters'\n    penguins = 'ice lovers'\n    birds    = 'fluffy soft dinosaurs'\n{: .language-ruby include=\"external_file.rb\"}\n\n
```

This is close but it’s missing the code block tags. This could be for one of two reasons;

1. Kramdown prefers indented blocks of code and it’s stripping them out in the super method of `convert_codeblock`
2. We are replacing it when we push the file contents onto el.value.

To test this, let’s remove the grabbing of files from the method so it looks like:

{:lang=”ruby”}
def convert_codeblock(el, indent)
return super(el, indent)
end

Then we can test again:

{:lang=”ruby”}
require ‘./test_md_render.rb’

Kramdown::Document.new(IO.read(‘./test.md’)).to_includey

We get back:

{:lang=”md”}

```
# Heading\n\nParagraph\n\nCode sample\n\n    nemo = 'missing'\n{: .language-ruby
# include=\"external_file.rb\"}\n\n
```

This means Kramdown is stripping it out. We have a few options;

1. Live with this format.
2. Override the parent method.
3. Look for a configuration option.

Living with it is easy and can be covered in a single 300 page book called “Living with it”. Coming to Oprah’s reading
book club this fall.

I digress. Let’s take a look at the source for the super function in Kramdown:

{:lang=”ruby”}
def convert_codeblock(el, opts)
el.value.split(/\n/).map {|l| l.empty? ? “ “ : “ #{l}”}.join(“\n”) + “\n”
end

Looks like the value is split on and then some padding inserted. This means the tags for the element, the three `~`
thingies, they are not used at all. We can override and add them back by doing:

{:lang=”ruby”}
def convert_codeblock(el, indent)
if el.attr.include?(‘include’)
file_path = File.expand_path(Dir.pwd + ‘/‘ + el.attr[‘include’])
el.value = IO.read(file_path) if File.exists?(file_path)
end

```
  el.value.split(/\n/).map {|l| l.empty? ? "    " : "    #{l}"}.join("\n") + "\n"
end
```

Run it like this:

{:lang=”ruby”}
require ‘./test_md_render.rb’
s = Kramdown::Document.new(IO.read(‘./test.md’)).to_includey
File.write(‘out.md’, s)

Open up `out.md` and you should have:

{:lang=”md”}

```
# Heading

Paragraph

# Code sample

~~~ ruby
cats     = 'dog haters'
penguins = 'ice lovers'
birds    = 'fluffy soft dinosaurs'
~~~
{: .language-ruby include="external_file.rb"}
```

We are pretty much good to go now. Have fun!

Including Source Files With Pandoc

Pandoc doesn’t have the ability to include source files built in but it’s fairly easy to add.
Just use the following code:

{:lang=”haskell”}
— includes.hs
import Text.Pandoc

```
doInclude :: Block -> IO Block
doInclude cb@(CodeBlock (id, classes, namevals) contents) =
  case lookup "include" namevals of
       Just f     -> return . (CodeBlock (id, classes, namevals)) =<< readFile f
       Nothing    -> return cb
doInclude x = return x

main :: IO ()
main = getContents >>= bottomUpM doInclude . readMarkdown def
                   >>= putStrLn . writeMarkdown def
```

The syntax to make use of this looks like this:

{:lang=”md”}
{include=”external_file.rb”}
this will be replaced by external_file.rb

That is it.

Including Files via LaTeX

In principle, this is quite straightforward and involves a single command, but in practice it’s not quite so simple.

First, we need to install the *listings* package. If you’re on Ubuntu do the following:

{:lang=”sh”}
$ sudo apt-get install texlive-extra-utils

Let’s test to see if is working. Do the following:

{:lang=”sh”}
$ mkdir test_text_including_external_files
$ cd test_text_including_external_files
$ touch test_include_external_file.tex
$ touch external_file.rb

Add some contents to our external ruby file:

{:lang=”ruby”}
god = []
rainbows = nil

```
rainbows = 'show' if god.include?('anger')

longrubylinesolongitwhaswrappingissues = 'Not rapping issues but wrapping issues. Like if Santa had issues you know'
```

Now add the following to the tex file:

{:lang=”tex”}
\usepackage{listings}
\lstinputlisting{external_file.rb}

Open it with gummi:

{:lang="sh"}
```
$ gummi test_include_external_file.tex
```

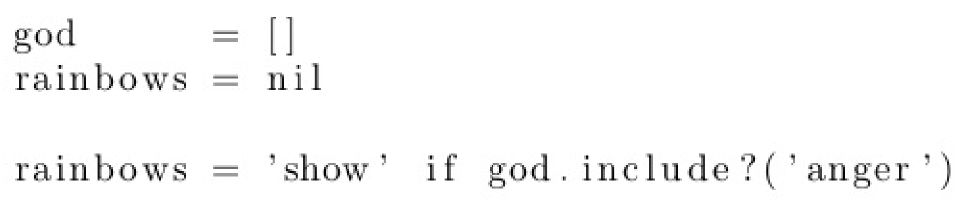
Now we get an error page in Gummi. Click on the build log (should also be in stdout if you ran gummi through terminal)
tab and you should see:

{:lang="sh"}
```
(/usr/share/texlive/texmf-dist/tex/latex/base/article.cls
Document Class: article 2007/10/19 v1.4h Standard LaTeX document class
(/usr/share/texlive/texmf-dist/tex/latex/base/size10.clo))
(/tmp/gummi_S8ZDXW.aux)
[1{/var/lib/texmf/fonts/map/pdftex/updmap/pdftex.map}]
(/tmp/gummi_S8ZDXW.aux) )
Output written on /tmp/gummi_S8ZDXW.pdf (1 page, 38476 bytes).
Transcript written on /tmp/gummi_S8ZDXW.log.
```

This is telling us that we need declare a document class first. Let’s do that:

{:lang="tex"}
```
\documentclass{article}
\usepackage{listings}
\begin{document}
\lstinputlisting{external_file.rb}
\end{document}
```

We get:



```
god      = []
rainbows = nil

rainbows = 'show' if god.include?('anger')
```

{:center=""}

It would be nice to get some syntax highlighting. We can get this by using Minted. Minted is a package for LaTeX that
wraps pygments and makes it available to LaTeX. First we need to make sure we have pygments installed.

A> If you think you have pygments installed but are not sure then run:
A>
A> {:lang="sh"}
A> $ pygmentize

You can install pygments by doing the following:

On Ubuntu

```
{:lang="sh"}
$ sudo apt-get install python-pygments
```

On Mac

```
{:lang="sh"}
$ sudo easy_install Pygments
```

After you have installed create a file called minted.tex:

```
{:lang="sh"}
$ touch minted.tex
```

To use *minted* do the following:

```
{:lang="tex"}
\documentclass{article}
\usepackage{minted}
\begin{document}
\inputminted{ruby}{external_file.rb}
\end{document}
```

Open with gummi and see the results:

```
{:lang="sh"}
$ gummi minted.tex
```

We get back an error image. If we view the build log we can see the problem:

```
{:lang="sh"}
This is pdfTeX, Version 3.1415926-2.4-1.40.13 (TeX Live 2012/Debian)
\write18 enabled.
entering extended mode
(./.minted.tex.swp
LaTeX2e <2011/06/27>
Babel and hyphenation patterns for english, dumylang, nohyphenation, lo
```

aded.
(/usr/share/texlive/texmf-dist/tex/latex/base/article.cls
Document Class: article 2007/10/19 v1.4h Standard LaTeX document class
(/usr/share/texlive/texmf-dist/tex/latex/base/size10.clo))

```
! LaTeX Error: File `minted.sty' not found.

Type X to quit or <RETURN> to proceed,
or enter new name. (Default extension: sty)

Enter file name:
./.minted.tex.swp:3: Emergency stop.
<read *>

l.3 \begin
          {document}^^M
./.minted.tex.swp:3:  ==> Fatal error occurred, no output PDF file produced!
Transcript written on /tmp/.minted.tex.log.
```

The key line is:

{:lang=”sh”}
! LaTeX Error: File `minted.sty’ not found.

This is telling us that minted is not installed. Let’s try to install via apt, because I’m lazy and you should be to;
laziness = productiveness.

We can search for the minted package by doing:

{:lang=”sh”}
$ apt-cache search minted

We get back the following:

{:lang=”sh”}
texlive-latex-extra - TeX Live: LaTeX supplementary packages

Bingo! Or as the IT kids say these days, “0118 999 881 999 119 725 3”!
Let’s install it using the following lines of
magic:

{:lang=”sh”}
$ sudo apt-get install texlive-latex-extra

Once installed we can run gummi again:

{:lang=”sh”}
$ gummi minted.tex

And finally we get back: !(External File Code Block Highlighting With Wrapping Issues](images/ExternalFilesLaTeXMintedWrappingIssues.png)
{:center=””}

You may notice a problem that I have so far glossed over, line wrapping issues. Unfortunately, there is a not solution
to this without some heavy hacking of minted. Instead we have two options;

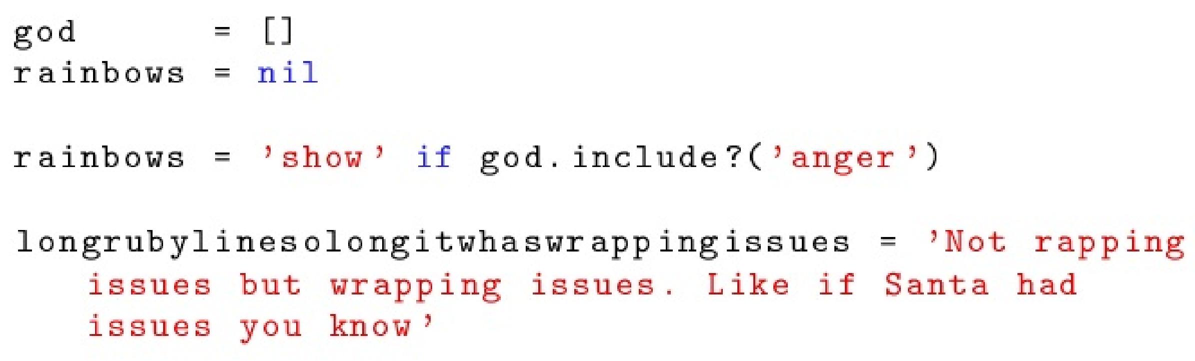
1. We can just manually wrap lines inserting a good old `\` or just by re-organizing until the
   line is shorter.
2. We can customize *Listings* to use colors.

Let’s do the latter. We have to start by configuring Listings which we do with the `lstset` macro.

{:lang=”tex”}
\documentclass{article}

```
\usepackage[T1]{fontenc}

\usepackage{color}
\definecolor{bluekeywords}{rgb}{0.13,0.13,1}
\definecolor{greencomments}{rgb}{0,0.5,0}
\definecolor{redstrings}{rgb}{0.9,0,0}

\usepackage{listings}
\lstset{language=Ruby,
  breaklines=true,
  showstringspaces=false,
  commentstyle=\color{greencomments},
  keywordstyle=\color{bluekeywords},
  stringstyle=\color{redstrings},
  basicstyle=\ttfamily
}

\begin{document}
\lstinputlisting{external_file.rb}
\end{document}
```

What we have done is set some colors for things like keywords, strings and comments. This results in the following:



```
god        = []
rainbows = nil

rainbows = 'show' if god.include?('anger')

longrubylinesolongitwhaswrappingissues = 'Not rapping
    issues but wrapping issues. Like if Santa had
    issues you know'
```

{:center=""}

How do we add this to our output in a Markdown to LaTeX workflow though? There are a few different ways but the
simplest and the one keeping most in line with our way of doings things so far is to use a template. Instead of passing
`--template document` to kramdown we will create our own and pass the full file path in the arguments.

Let's open `kramdown/data/kramdown/document.latex` . We will be presented with a relatively simple erb template

{:lang="erb"}
<%
encmap = {
'UTF-8' => 'utf8x',
'US-ASCII' => 'ascii',
'ISO-8859-1' => 'latin1',
'ISO-8859-2' => 'latin2',
'ISO-8859-3' => 'latin3',
'ISO-8859-4' => 'latin4',
'ISO-8859-5' => 'latin5',
'ISO-8859-9' => 'latin9',
'ISO-8859-10' => 'latin10',
'CP850' => 'cp850',
'CP852' => 'cp852',
'CP858' => 'cp858',
'CP437' => 'cp437',
'CP865' => 'cp865',
'CP1250' => 'cp120',
'CP1252' => 'cp1252',
'CP1257' => 'cp1257'
}
%>

```
\documentclass{scrartcl}
<% if RUBY_VERSION >= ‘1.9’ %>
\usepackage[<%= encmap[@body.encoding.name] %>]{inputenc}
<% else %>
\usepackage[mathletters]{ucs}
\usepackage[utf8x]{inputenc}
<% end %>
\usepackage[T1]{fontenc}
\usepackage{listings}
<% @converter.data[:packages].each {|pkg| %>\usepackage{<%= pkg %>}
<% } %>
\usepackage{hyperref}
```

```
<% if @converter.data[:packages].include?('fancyvrb') %>
\VerbatimFootnotes
<% end %>

\hypersetup{colorlinks=true,urlcolor=blue}

\begin{document}
<%= @body %>
\end{document}
```

We are already including listings so all we need to do is add `lstset` after the `usepackages` like this:

{:lang=”erb”}
….

```
<% if @converter.data[:packages].include?('fancyvrb') %>
\VerbatimFootnotes
<% end %>

\lstset{language=Ruby,
  breaklines=true,
  showstringspaces=false,
  commentstyle=\color{greencomments},
  keywordstyle=\color{bluekeywords},
  stringstyle=\color{redstrings},
  basicstyle=\ttfamily
}
```

Save this someplace and then pass it to Kramdown with the template argument. For example;

{:lang=”sh”}
$ kramdown –template customtemplate.erb.latex

That will do it. Enjoy!

Generating a Table of Contents

Generating a table of contents in Kramdown is straightforward but there are a few practical problems you can run into.
Before we examine how it works in Kramdown let’s take a look at how it works as LaTeX.

I> You’ll have to build the PDF twice for the Table of Contents Page to appear. Remember, it takes multiple passes for
I> LaTeX to build up its output. One pass to get all the sections and headings then another to generate it.

Open a generated Joined.tex and you should see something like:

{:lang=”tex”}
\documentclass{scrartcl}

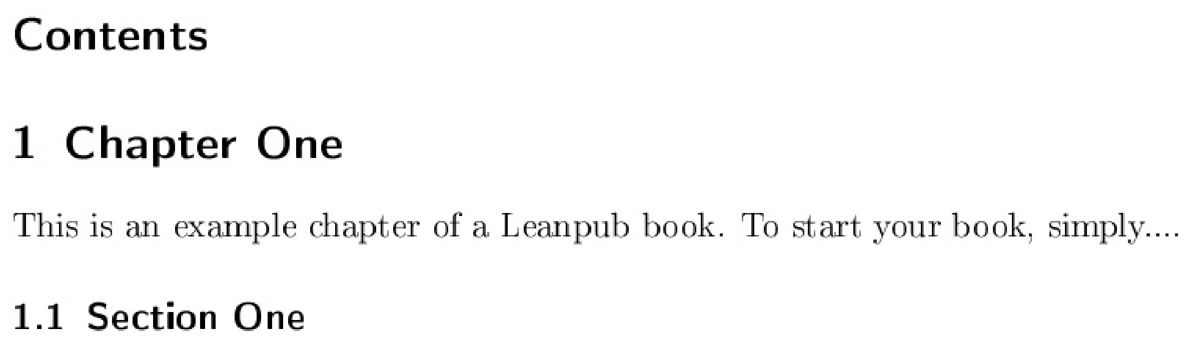
```
\usepackage[utf8x]{inputenc}

\usepackage[T1]{fontenc}
\usepackage{listings}
\usepackage{graphicx}

\usepackage{hyperref}

\hypersetup{colorlinks=true,urlcolor=blue}

\begin{document}
\section{Chapter One}\hypertarget{chapter-one}{}\label{chapter-one}

This is an example chapter of a Leanpub book. To start your book, simply....

\subsection{Section One}\hypertarget{section-one}{}\label{section-one}

\end{document}
```

Adding a table of contents page is straightforward, we just need to use the relevant command. It looks like this:

{:lang="tex"}
\begin{document}
\tableofcontents
…

I'm not skimping on commands here, all you need to do is call `\tableofcontents` to get a generated TOC. LaTeX will
build up its table of contents after parsing all the sections, and then it will insert the table of contents where the
command is called.

Running pdflatex against this results in the following PDF:


Contents

1 Chapter One

This is an example chapter of a Leanpub book. To start your book, simply....

1.1 Section One

{:center=""}

Of course we cant be manually inserting this command every time we regenerate LaTeX. We are going to have to get
Kramdown to do it for us. Kramdown handles table of contents without any modifications. How do we generate a TOC page
though? Let's find out by opening `kramdown/converter/latex.rb`. Searching for *\tableofcontents*
reveals that the TOC is generated when the appropriate options are set:

{:lang="ruby}
if !@data[:has_toc] && (el.options[:ial][:refs].include?('toc') rescue nil)

This is telling us that if we add an attribute called *toc* to a list it will be replaced with the
call to `\tableofcontents`. For example:

{:lang="md"}

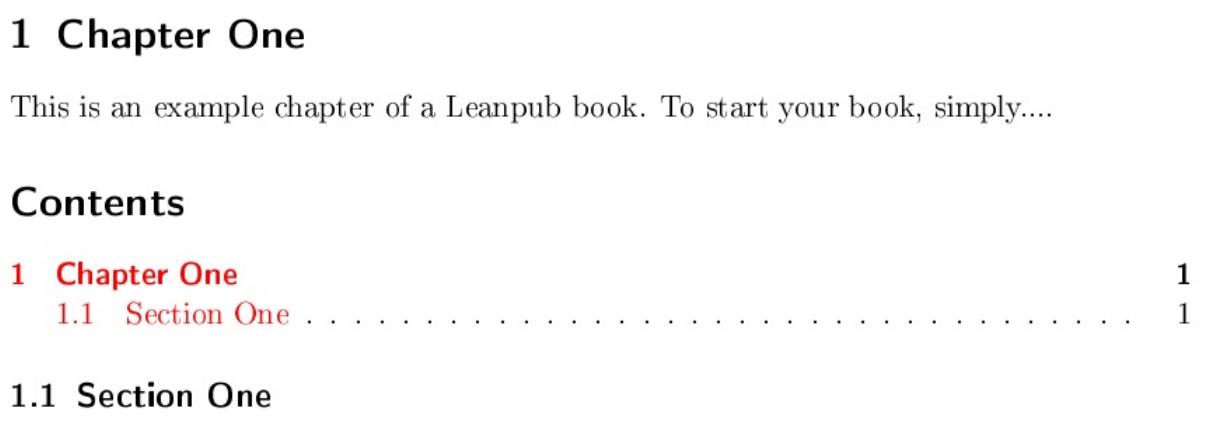
```
* Will be replaced with the ToC, excluding the "Contents" header
{:toc}
```

Running this through Kramdown results in the following LaTeX:

{:lang="tex"}
\documentclass{scrartcl}

```
\usepackage[utf8x]{inputenc}

\usepackage[T1]{fontenc}
\usepackage{listings}
\usepackage{graphicx}

\usepackage{hyperref}

\hypersetup{colorlinks=true,urlcolor=blue}

\begin{document}
\section{Chapter One}\hypertarget{chapter-one}{}\label{chapter-one}

This is an example chapter of a Leanpub book. To start your book, simply....

\tableofcontents
\subsection{Section One}\hypertarget{section-one}{}\label{section-one}

\end{document}
```

And the following PDF:



1 Chapter One

This is an example chapter of a Leanpub book. To start your book, simply....

Contents

1 Chapter One 1
1.1 Section One 1

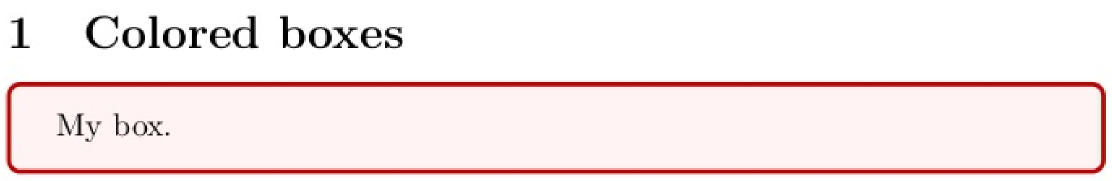
1.1 Section One

{:center=""}

That is it!

Custom thing a ma bobs.

You've seen them before, the custom little framed boxes of content. Maybe they look this:


1 Colored boxes

My box.

{:center=""}

Now you're thinking, I'd really like to build some boxes like that; to create notes, colored backgrounds and little
icons. How do we do this on the limited Markdown language spec? The answer is extensions. The syntax for Markdown
extensions uses an opening and ending tag that looks like this:

```
{:lang="md"}
{::comment}
This is a comment which is
completely ignored.
{:/comment}
```

We need to do one thing to make this flexible enough for our needs. We need to modify the *handle_extension* method in
kramdown/parser/kramdown/extensions.rb so it handles any named extension.

Change the last else in the method to the following:

```
{:lang="ruby"}
else
@tree.children << Element.new(name.to_sym, body, nil, :category => type)
if body.kind_of?(String)
true
end
```

Then we can just define the method on our converter class:

```
{:lang="ruby"}
def convert_comment(el, opts)
el.value.split(/\n/).map {|l| "% #{l}"}.join("\n") + "\n"
end
```

The first thing we need is the LaTeX for creating these boxes. We are going to use a package aptly called `tcolorbox` .
The syntax is rather simple and looks like:

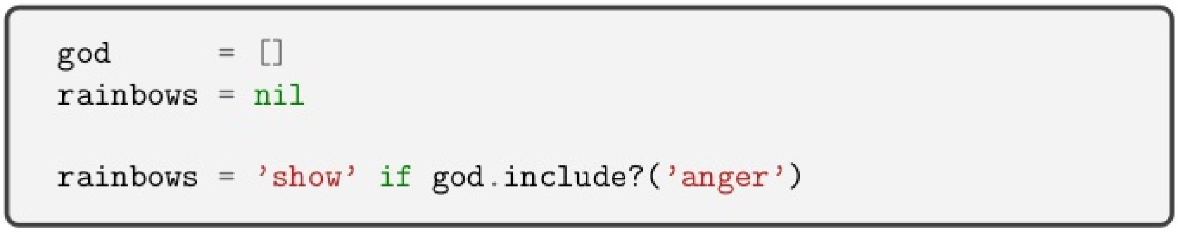
{:lang="tex"}
\begin{tcolorbox}[options]
My box.
\end{tcolorbox}

We will simply use the default options for now. Let's create a simple example document that loads some code up into a
box:

{:lang="tex"}
\documentclass{article}
\usepackage{tikz,lipsum,minted}
\usepackage[listings,theorems,skins]{tcolorbox}

```
\begin{document}
\begin{tcolorbox}
  \inputminted{ruby}{external_file.rb}
\end{tcolorbox}
\end{document}
```

Saving and running that doc through pdflatex results in the following:



```
god       = []
rainbows  = nil

rainbows  = 'show' if god.include?('anger')
```

{:center=""}

If you need line breaks you can do:

{:lang="tex"}
\documentclass{article}
\usepackage{tikz,lipsum}
\usepackage[listings,theorems,skins]{tcolorbox}

```
\usepackage[T1]{fontenc}

\usepackage{color}
\definecolor{bluekeywords}{rgb}{0.13,0.13,1}
\definecolor{greencomments}{rgb}{0,0.5,0}
\definecolor{redstrings}{rgb}{0.9,0,0}

```

```
\usepackage{listings}
\lstset{language=Ruby,
  breaklines=true,
  showstringspaces=false,
  commentstyle=\color{greencomments},
  keywordstyle=\color{bluekeywords},
  stringstyle=\color{redstrings},
  basicstyle=\ttfamily
}

\begin{document}
\begin{tcolorbox}
\lstinputlisting{external_file.rb}
\end{tcolorbox}
\end{document}
```

That is it for the LaTeX bit. Now for the Markdown bit.

We need a Markdown extension that generates this, let’s call it *tcolorbox*. The Markdown to use it would then look
like this:

{:lang=”md”}
{::tcolorbox}
code here
{:/tcolorbox}

We only have one problem to resolve, how do we support any type of content in our blocks? The answer is simple, we just
parse what is in the block as a separate document and run *to_latex* on it. This allows us to put any arbitrary
Markdown we want in the block.

The code for this is rather simple:

{:lang=”ruby”}
def convert_tcolorbox(el, opts)
@data[:packages] << ‘tcolorbox’ unless @data[:packages].include?(‘tcolorbox’) # Add the package

“\begin{tcolorbox} #{: :Document.new(el.value).to_latex}
\end{tcolorbox}”
end

Now we add this method to our Kramdown fork and re-build/re-install the gem.

When we run a test document:

{:lang=”md”}
{::tcolorbox}

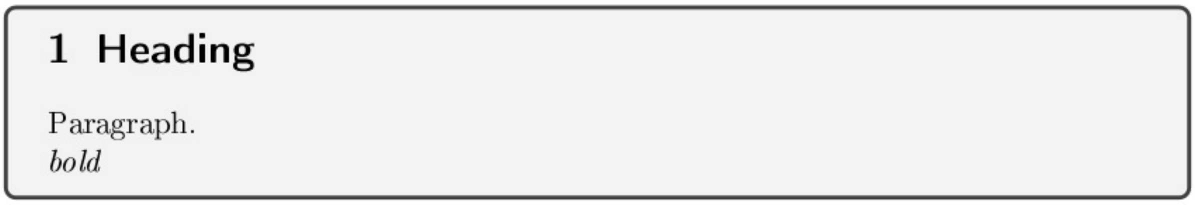
```
# Heading

Paragraph.

*bold*
{:/tcolorbox}
```

We get the following LaTeX:

{:lang=”tex”}
\documentclass{scrartcl}

```
\usepackage[utf8x]{inputenc}

\usepackage[T1]{fontenc}
\usepackage{listings}
\usepackage{tcolorbox}

\usepackage{hyperref}

\hypersetup{colorlinks=true,urlcolor=blue}

\begin{document}
\begin{tcolorbox} \section{Heading}\hypertarget{heading}{}\label{heading}

Paragraph.

\emph{bold}

 \end{tcolorbox}
% This is a comment which is
% completely ignored.

\end{document}
```

And the resulting PDF is:



1 Heading

Paragraph.
bold

{:center=""}

A key binding awaits

The days before `<kbd>` were a dark time for the web. It was a time without clear shortcuts. A time when keys went
unstyled and blended into the text, never to be seen, like a secret agent behind a cactus. It was a time when
instructions where easily missed and humans were easily frustrated. Then came the kbd and the presentation of
shortcuts was changed for ever.

No longer were replies to questions limited by the styles of plain text. The world had
a new tag and a whole new world was opened up to it.

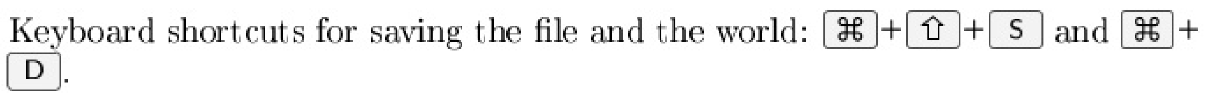
Heading back to the dark ages for PDF's sake is not an option. We have to bring the future to PDFs. How do we do this?
First things first, we have to know the LaTeX it takes to build this. Thankfully, there is a package called *menukeys*
that will handle all our kbd needs.

We use menukeys like this:

{:lang="tex"}
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage{menukeys}

```
\begin{document}
Keyboard shortcut for saving the file and the world: \keys{\cmd+\shift+S}
\end{document}
```

This results in the following output:


Keyboard shortcuts for saving the file and the world: ⌘+⇧+S and ⌘+D.

{:center=""}

Now we need to add support for kbd to our Markdown. Markdown supports html tags so all we have to do is just use the
kbd tag and then transform it into the relevant LaTeX. We could invent our own tag but using the html one makes it
easy to support html and thus html based ebook formats.

Add the following Markdown to a test doc:

{:lang="md"}
Keyboard shortcut for saving the file and the world: cmd + shift + S

When we run Kramdown against this Markdown we will get the following error:

{:lang="sh"}
Warning: Can't convert HTML element

If we search for *"Can't convert HTML element"* in *kramdown/converter/latex.rb* we will find a method called *convert_html_element*:

{:lang="ruby"}
def convert_html_element(el, opts)
if el.value == 'i'
"\emph{#{inner(el, opts)}}"
elsif el.value == 'b'
"\emph{#{inner(el, opts)}}"
else
warning("Can't convert HTML element")
''
end
end

Adding support for *kbd* should be rather straightforward. We will need an if statement and something that appends the
menukeys package to `@data[:packages]` .

The following should do:

```
{:lang=”ruby”}
elsif el.value == ‘kbd’
@data[:packages] << ‘menukeys’ unless @data[:packages].include?
(‘menukeys’) # Add the package
“\keys{#{inner(el, opts)}}”
```

Make the modification to a Kramdown fork and then run:

```
{:lang=”sh”}
$ rake install
$ rbenv rehash # Optional use only if rbenv is installed
```

to update your Kramdown binary in rubygems.

That is it!

Krambook: Build a Gem

- Krambook: Build a Gem

Krambook: Build a Gem

It’s time to package up all these custom Kramdown extensions into a Gem.

T> If you so desire you can skip this section and simply visit the repo and
T> install the completed gem.

First create a directory for the gem:

{:lang=”sh”}
$ mkdir krambook
$ cd krambook

Now we need to add a gemspec file:

{:lang=”sh”}
$ touch krambook.gemspec

Add the contents:

{:lang=”ruby”}

```ruby
#!/usr/bin/env gem build
# encoding: utf-8

Gem::Specification.new do |s|
  s.name                       = 'krambook'
  s.rubyforge_project          = 'krambook'
  s.authors                    = ["K-2052"]
  s.email                      = 'k@2052.me'
  s.summary                    = 'Kraft you ebooks with Kramdown.'
  s.homepage                   = 'http://github.com/bookworm/krambook'
  s.description                = 'Kraft you ebooks with Kramdown. Intended as a collection
                                  of helpers for working with Markdown based ebooks.
                                  Currently operates like a local version of LeanPub.'
  s.required_rubygems_version = '>= 1.3.6'
  s.version                    = '0.0.1'
  s.date                       = Time.now.strftime("%Y-%m-%d")

  s.files             = `git ls-files`.split("\n") | Dir.glob("{lib}/**/*")
  s.test_files       = `git ls-files -- {test,spec,features}/*`.split("\n")
  s.executables      = `git ls-files -- bin/*`.split("\n").map{ |f| File.basename(f) }
  s.require_paths    = ['lib']
  s.rdoc_options     = ['--charset=UTF-8']

  s.add_dependency('kramdown')
  s.add_dependency('bundler', '~> 1.0')
  s.add_dependency('thor')
  s.add_dependency('active_support')
end
```

Most of these lines are self explanatory. We use git to pull our files and we add *Thor* as a dependency. Thor is a
wrapper around command line and file generation stuff. Think rake, but on steroids. Thor will allow us to construct
minimal code for handling arguments.

Add a Gemfile for bundler’s sake:

{:lang=”ruby”}
source ‘https://rubygems.org‘

```
gemspec
```

This tells bundler to use the gemspec as a source for the gems. Run `$ bundle install` to make sure things are
installed.

Now we need to a base file to load up our gem file files when someone does `require 'krambook'` :

{:lang=”sh”}
$ mkdir lib
$ cd lib
$ touch krambook.rb

Open up `krambook.rb` and define a class for the CLI interface:

{:lang=”ruby”}
require ‘kramdown’
require ‘thor’

```
module Krambook
  class CLI < Thor
    include Thor::Actions

    class_option :type, :desc => 'The type to convert to', :aliases => '-t', :default => 'kramdown', :type => :string
    class_option :help, :type => :boolean, :desc => 'Show help usage'
    default_task :join

    def join
    end
  end
end
```

Now on to our convert method. We need to add the code we created earlier and modify it to to use arguments passed to it.

{:lang=”ruby”}
def join
files = []
joined_file_path = File.expand_path ‘./manuscript/‘ + ‘Joined.md’
File.delete(joined_file_path) if File.exist?(joined_file_path)

```
  File.open(File.expand_path('./manuscript/Book.txt'), 'r') do |f|
    f.each_line do |line|
      files << line.gsub(/\s+/, "")
    end
  end

  File.open(joined_file_path, 'a') do |f|
    files.each do |file_path|
      full_file_path = File.expand_path './manuscript/' + file_path
      f.puts Kramdown::Document.new(IO.read(full_file_path)).send("to_#{options[:format]}".to_sym) if File.exists?(full_file_path)
      f.puts '' unless file_path = files.last
    end
  end
end
```

Let’s add that external file inclusion converter we created earlier to `lib/kramdown/converter` :

{:lang=”ruby”}
module Kramdown
module Converter

class Includey < ::Kramdown
def convert_codeblock(el, indent)
if el.attr.include?(‘include’)

file_path = File.expand_path(Dir.pwd + ‘/‘ + el.attr[‘include’])
el.value = IO.read(file_path) if File.exists?(file_path)
end

```
        attr = el.attr.dup
        lang = extract_code_language!(attr)
        "~~~ #{lang}\n" + el.value.split(/\n/).map {|l| l.empty? ? "" : "#{l}"}.join("\n") + "\n~~~\n"
      end
    end
  end
end
```

Remember to require it at the top of `lib/krambook.rb`

{:lang=”ruby”}
require ‘kramdown’
require ‘kramdown/converter/includey’

We want this to work from CLI so do the following.

{:lang=”sh”}
$ mkdir bin
$ touch bin/krambook
$ chmod a+x bin/krambook

Then add the following to *bin/krambook*:

{:lang=”ruby”}

```
#!/usr/bin/env ruby

$LOAD_PATH.unshift File.join(File.dirname(__FILE__), '..', 'lib')

require 'krambook'
Krambook::CLI.start
```

We can test this by running the bin file in a directory with a book. For example:

{:lang=”sh”}
$ ruby /home/k2052/creations/krambook/bin/krambook

We get the following back:

{:lang=”sh”}

```
[WARNING] Attempted to create command “convert” without usage or
description. Call desc if you want this method to be available as command
or declare it inside a no_commands{} block. Invoked from

“/home/k2052/creations/krambook/lib/krambook.rb in `‘“.
Could not find command “”.
```

We can fix both issues by adding a desc before the join method definition.

Add the following to `krambook.rb` :

```
{:lang=”ruby”}
desc ‘join’, ‘Joins the markdown files’
def join
```

Run it again and you should have a *Joined.md’* in the manuscript folder.
Now it’s just a matter of submitting our gem
to rubygems.

Note: Obviously you don’t actually perform this step as I’m the maintainer of the Krambook gem.

First add a README.md file:

```
{:lang=”sh”}
$ touch README.md
```

Then do all the git stuff:

```
{:lang=”sh”}
$ git init .
$ git add .
$ git commit -m “first commit” -a
```

If your bin file is ignored you may have to do:

```
{:lang=”sh”}
$ gi]t add bin/ —force
```

Then we need to build the gem and submit:

```
{:lang=”sh”}
$ gem build krambook.gemspec
$ gem push krambook-0.0.1.gem
```

Done!

A Sublime Markdown Experience

A Sublime Markdown Experience

If you write a book in Markdown chances are very high that you are going to write a lot of Markdown. Well, unless you
have hired one of those ghost Markdown writers or made a deal with a Markdown demon. It is going to be helpful to tweak
your environment to be specific to Markdown. Thankfully Sublime Text 2 is flexible enough that it can quickly become
the best friend of any Markdown writer.

Count my chars, measure my words, tell me the lines, express my worth

Word count is a powerful tool for judging both the completion of your work and to measure your self worth as a writer.
Are you procrastinating? Are you zooming along like a super human Stephen King? Only word count can tell you. Well,
that, and the quality of your writing. But who needs quality writing when you have measurements like word count.

Adding word counts to Sublime Text 2 is quite easy, just Install the package WordCount.
Once installed, you will have a word count in the status bar below your files. Now drive up that count!

Getting Sublime

I wish this section was longer and reaching Nirvana for Markdown editing was harder that way I could elevate my word
count. Unfortunately, it’s as simple as installing two extensions;

SmartMarkdown and
MarkdownEditing. Together, these two will add everything you need. *SmartMarkdown* will handle
things like smart folding, adjusting heading levels etc. *MarkdownEditing* will handle all your nifty shortcuts for
inserting Markdown things.

Not to worry though, I can up my word count a bit with some configuration suggestions.

You are going to want to use a better theme for Markdown editing, one that gives you nice highlighting for everything,
one that is designed for writing not coding. MarkdownEditing comes with an excellent theme inspired by Byword that
looks very nice. You can enable it for Markdown files by adding it to your *MultiMarkdown.sublime-settings* and
Markdown.sublime-settings files like so:

{:lang="json"}
{
“color_scheme”: “Packages/MarkdownEditing/MarkdownEditor-Focus.tmTheme”
}

You might also want to enable Markdown syntax on `.txt` files for when you’re utilizing Leanpub. You can force Leanpub
to use `.md` but you might be the type of person that likes `.txt` to be treated as Markdown. You can do this easily by
adding the following to *Markdown.sublime-settings*:

{:lang="json"}
{
“extensions”:
[
“txt”
]
}

You might also want to try out the ApplySyntax plugin which will handle detecting syntax based on the contents of a
file. Get it via package control or Github

Bugs with MarkdownEditing

There is a bug in MarkdownEditing on installations without the Menlo font. The font is set in the settings for the
syntax manually. Doing this causes bizarre issues, like rulers disappearing and fonts not resizing. You can fix it by
resetting your preferred font in the MultiMarkdown .sublime-settings and Markdown.sublime-settings files:

{:lang=”json”}
{
“font_face”: “Menlo”
}

Font settings being set in a Package also causes issues with using commands for increasing/decreasing text size. You
can fix this by commenting out `font_size` in `MarkdownEditing packages folder/Markdown.sublime-settings` (click browse
packages to find it).

May your Markdown experience be Sublime!

All the features of Leanpub

All the features of Leanpub

Leanpub has a few nifty doodads that will be useful in the generation of our documents. Things like; code includes,
asides, warning blocks, info blocks and death rays. With the exception of that last one (who needs a death ray when you
have a… I've said too much.) they're all extremely useful. If you use Leanpub to publish your book in the early
phases then you might become dependent on them and need to replicate them or risk a terrible 48 hours of detoxing in a
dark room. Not to worry, you can replicate Leanpub features as easily as Tyrell Corp replicates.

Move Aside

Most of the Leanpub extensions to Kramdown utilize angle brackets to indicate things. If you want a sidebar you'd do

{:lang="md"}
A> This is a sentence or in an aside

Or if you wanted a warning box with an icon you'd do:

{:lang="md"}
W> This is a warning about the end of the world and the beginning of whimpers

To replicate these type of *tags* we will need two things;

1. A parser for the tag.
2. A convert method that takes that block and turns it into an output string.

Let’s look at a parser to see how one is built. Open up `kramdown/parser/kramdown/blockquote.rb` :

{:lang=”ruby”}
BLOCKQUOTE_START = /^#{OPT_SPACE}> ?/

```ruby
# Parse the blockquote at the current location.
def parse_blockquote
  result = @src.scan(PARAGRAPH_MATCH)
  while !@src.match?(self.class::LAZY_END)
    result << @src.scan(PARAGRAPH_MATCH)
  end
  result.gsub!(BLOCKQUOTE_START, '')

  el = new_block_el(:blockquote)
  @tree.children << el
  parse_blocks(el, result)
  true
end
define_parser(:blockquote, BLOCKQUOTE_START)
```

The parser defines itself by passing a name and Regex to the `define_parser` method. The Regex (`BLOCKQUOTE_START`)
simply looks for a space (optional) followed by a right angle bracket and another space which is made optional by the
`?` mark. The parser class iterates over defined parsers and when it reaches something that matches `BLOCKQOUTE_START`
it passes off the parsing to the matching method.

The parser method turns the content of the block into an element and then returns true to the parser
so it knows the element has been added to the element tree. It also moves a string scanner along until all the content for the block is parsed so the parser doesn’t hit this block again.

This is typical behavior for most parsers. Most parsers consist of two main parts;

1. A string scanner that moves along parsing things using simple matching.
2. Something that turns the strings into an element tree (AST) which is later converted to final output by dedicated output classes.

To accomplish the parsing bits for an aside we only need to duplicate the blockquote parser and tweak the Regex slightly. Create a new file called `kramdown/parser/aside.rb` with the following contents:

{:lang="ruby"}
require 'kramdown/parser/kramdown/blank_line'
require 'kramdown/parser/kramdown/extensions'
require 'kramdown/parser/kramdown/eob'

```ruby
module Kramdown
  module Parser
    class Kramdown

      ASIDE_START = /^#{OPT_SPACE}A> ?/

      # Parse the aside at the current location.
      def parse_aside
        result = @src.scan(PARAGRAPH_MATCH)
        while !@src.match?(self.class::LAZY_END)
          result << @src.scan(PARAGRAPH_MATCH)
        end
        result.gsub!(ASIDE_START, '')

        el = new_block_el(:aside)
        @tree.children << el
        parse_blocks(el, result)
        true
      end
      define_parser(:aside, ASIDE_START)

    end
  end
end
```

The only significant change is to the Regex. It now looks for an `A` fellowed by an angle bracket `>`.
The next step is to add a `convert_aside` to the HTML and LaTeX converter classes. Let's start with the HTML
conversion.

First we need to determine what Markup to utilize for an aside. We could easily craft this ourselves but we risk
running into compatibility issues with ePub readers. We should stick to what is known to work. We can open up the
generated HTML from [^export-html] Leanpub and pull the Markup and the CSS for asides.

The html for an aside is:

{:lang=”html”}

Now we just need to add a convert method that returns this. Open up `kramdown/converter/html.rb` and add a method called `convert_aside` :

{:lang=”ruby”}
def convert_aside(el, indent)
end

We still need to do two things;

1. Wrap the body in the markup.
2. Make sure we parse the contents of the block so we can use things like codeblocks inside an aside.

First step:

{:lang=”ruby”}
result = “

”
result << ‘
’

We now have a wrapper div. Now let’s add a call to inner() to pull the content:

{:lang=”ruby”}
result = “

”
result << inner(el, indent)
result << ‘
’

The final method now looks like:

{:lang=”ruby”}
def convert_aside(el, indent)
result = “

”
result << inner(el, indent)
result << ‘

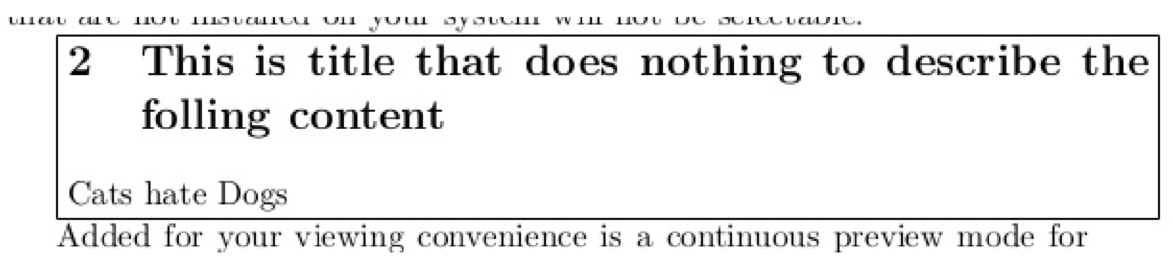
‘

end

That does it for the html portion. Now how do we get asides in our PDFs? We will need a LaTeX package that
constructs something similar. There is package called fancybox that will allow us to construct simple boxes that very closely resemble the Leanpub asides. The LaTeX for it looks like this:

```
{:lang=”TeX”}
\fbox{%
\begin{minipage}{\textwidth}
\section{This is title that does nothing to describe the folling content}
Cats hate Dogs
\end{minipage}}
```

This results in a box that looks like this:



that are not installed on your system will not be selectable.

2 This is title that does nothing to describe the folling content

Cats hate Dogs

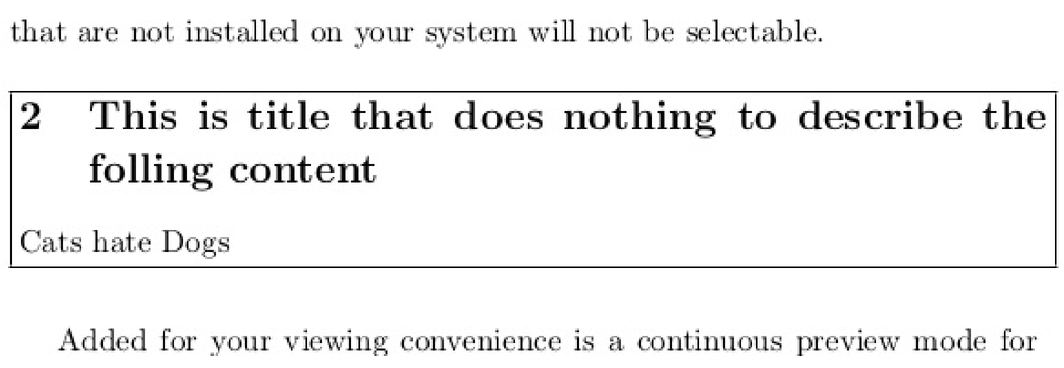
Added for your viewing convenience is a continuous preview mode for

{:center=””}

This is a bit tight and could use some padding and the left margin removed. We can add padding using `\vspace` and
remove margins using `\noident` . The LaTeX will then look like:

```
{:lang=”TeX”}
\vspace{5 mm}
\noindent
\fbox{%
\begin{minipage}{\textwidth}
\section{This is title that does nothing to describe the folling content}
Cats hate Dogs
\end{minipage}}
\vspace{5 mm}
```

When processed it looks like:


that are not installed on your system will not be selectable.

2 This is title that does nothing to describe the folling content

Cats hate Dogs

Added for your viewing convenience is a continuous preview mode for

{:center=""}

Now we just need to get this added to our LaTeX converter. Open up `kramdown/converter/latex.rb` and add a `convert_aside` method:

{:lang="ruby"}
```
def convert_aside(el, opts)
end
```

Append the package to packages:

{:lang="ruby"}
```
@data[:packages] << 'fancybox' unless @data[:packages].include?
('fancybox') # Add the package
```

Add the wrap string:

{:lang="ruby"}
```
result = <<-eos
\vspace{5 mm}
\noindent
\fbox{%
\begin{minipage}{\textwidth}
eos
```

Process the el's content through inner:

{:lang="ruby"}
```
result << inner(el, opts)
```

Then close the string:

{:lang="ruby"}
```
result << <<-eos
\end{minipage}}
\vspace{5 mm}
eos
```

The final method looks like:

{:lang="ruby"}
def convert_aside(el, opts)
@data[:packages] << 'fancybox' unless @data[:packages].include?
('fancybox') # Add the package
result = <<-eos
\vspace{5 mm}
\noindent
\fbox{%
\begin{minipage}{\textwidth}
eos

```
  result << inner(el, opts)

  result << <<-eos
    \end{minipage}}
    \vspace{5 mm}
  eos
end
```

[^export-html]:
To get html you can either utilize the export html feature (found in your book's actions tab) or
open the ePub (an ePub is actually just a zip of html files) and grab the HTML from there.

Icons

Leanpub provides warning, information, tips, error, discussion icon blocks etc. They all look pretty much the same and
consist of an icon on the left with some text on the right. They look like this:

W>## This is a Warning
W>
W> Warnings are generated by using `W>` at the beginning of lines.

All are implemented pretty much in the same manner, so we will only focus on implementing one: The information block.
First we need to open up some generated html and get the markup for it:

{:lang="html"}

```
      <td>
      </td>
    </tr>
  </tbody>
</table>
```

Next we will add a parser for it. Create a file called `infoblock.rb` in `kramdown/parser/kramdown` with the following contents:

{:lang="ruby"}
module Kramdown
module Parser
class Kramdown

```ruby
      INFOBLOCK_START = /^#{OPT_SPACE}I> ?/

      # Parse the aside at the current location.
      def parse_infoblock
        result = @src.scan(PARAGRAPH_MATCH)
        while !@src.match?(self.class::LAZY_END)
          result << @src.scan(PARAGRAPH_MATCH)
        end
        result.gsub!(INFOBLOCK_START, '')

        el = new_block_el(:infoblock)
        @tree.children << el
        parse_blocks(el, result)
        true
      end
      define_parser(:infoblock, INFOBLOCK_START)

    end
  end
end
```

Now open up `convert/html.rb` and add a method for converting an infoblock:

{:lang="ruby"}

def convert_infoblock(el, indent)
result = <<-eos

eos

```
  result << inner(el, indent)

  result << <<-eos
          </td>
        </tr>
      </tbody>
    </table>
  eos
end
```

If we want to do this for LaTeX we can utilize a package called *notes* that supports everything from information to warning notes. To construct an information note we use the following:

{:lang="TeX"}
\begin{informationnote}
Informative information about Penguins. Everything you need to know to survive an attack from an angry flightless bird.
\end{informationnote}

Now we just need to add this to the LaTeX converter. Open up `kramdown/converter/latex.rb` and add a `convert_infoblock` method:

{:lang="Ruby"}
def convert_infoblock(el, opts)
@data[:packages] << ‘notes’ unless @data[:packages].include?(‘notes’) # Add the package
result = ‘\begin{informationnote}’

```
  result << inner(el, opts)

  result << '\\end{informationnote}'
end
```

That is it!

You can repeat the process detailed in this section for any of the other types of note blocks.

Includes like Leanpub

We have already made a converter for including snippets but it just so happens that Leanpub has it’s own syntax for this. You can include a sample file like this:

{:lang=”md”}
<<(code/sample1.rb)

Replicating this syntax won’t be too hard we just leverage our previous code for including samples and combine it with a custom parser. First create a new file in `kramdown/parser/` and call it `includeblock.rb` . Place the following inside its bowels:

{:lang=”Ruby”}
require ‘kramdown/parser/kramdown/blank_line’
require ‘kramdown/parser/kramdown/extensions’
require ‘kramdown/parser/kramdown/eob’

```ruby
module Kramdown
  module Parser
    class Kramdown

      INCLUDEBLOCK_START = /^#{OPT_SPACE}<</
      INCLUDEBLOCK_MATCH = /^#{OPT_SPACE}<<\((.*)\)/

      # Parse the aside at the current location.
      def parse_includeblock
        result = @src.scan(INCLUDEBLOCK_MATCH)
        result.gsub!(INCLUDEBLOCK_START, '')
        result.gsub!('(', '')
        result.gsub!(')', '')

        el = new_block_el(:includeblock)
        @tree.children << el
        parse_blocks(el, result)
        true
      end
      define_parser(:includeblock, INCLUDEBLOCK_START)

    end
  end
end
```

Now open up `kramdown/converter/html.rb` and add a converter method for include blocks.

{:lang="Ruby"}
def convert_includeblock(el, indent)
file_path = File.expand_path(Dir.pwd + '/' + el.value)
el.value = IO.read(file_path) if File.exists?(file_path)
el.attr['class'] = "language-#{File.extname(file_path)}" if
File.exists?(file_path)

```
  convert_codeblock(el, indent)
end
```

This takes the file path; attempts to load the file if it exists, adds a language class based on the ext name and then finally passes the file contents off to `convert_codeblock` for processing.

Since we are utilizing `convert_codeblock` for the final transformation, the method is the exact same for the LaTeX converter. Just copy it on over.

Make the code blocks detect their language

You can make codeblocks detect their language from the lang attribute by overriding the `convert_codeblock` method and pulling the language from attribute and then inserting it into onto the class attribute:

{:lang="ruby"}
def convert_codeblock(el, indent)
el.attr['class'] = "language-#{el.attr['lang']} #
{el.attr['class']}" if el.attr.include?('lang')
super(el, indent)
end

Now your codeblocks will be properly highlighted.

Use pygments with Kramdown

This is really straightforward but I thought it worth including instructions and a snippet anyway.
Override the `convert_codeblock` method and change its contents to the following:

{:lang="ruby"}
def convert_codeblock(el, indent)
attr = el.attr.dup
lang = extract_code_language!(attr)
result = Pygments.highlight(code, :lexer => lang)

```
  "#{' '*indent}<div#{html_attributes(attr)}>#{result}#{' '*indent}
</div>\n"
end
```

That is it! I told you it was really straightforward.

W> Make sure you have required the *pygments* gem at some point.

Pitfalls

Pitfalls

Chances are you are gong to fall into a pit sooner or later. Holes abound in the land of code and Markdown. These
sections help you avoid frustration and increase the delight of crafting your ebook.

Markdown Issues

Markdown can go wrong, don’t let it go wrong on you. If it does go wrong, you can turn to the following headings for
help. Trust them, they’re headings.

Duplicate Link Issues

```
Warning: Duplicate link ID '1' - overwriting
```

This occurs when you have duplicate links in your Markdown. This can happen when you have two of the same links or a
link without a title. The solution is to use link references:

{:lang=”md”}
A [link][kramdown hp] to the homepage.

```
[kramdown hp]: http://kramdown.rubyforge.org "hp"
```

Footnote ID issues

```
Warning: Footnote marker '1' already appeared in document, ignoring newly found marker
```

To resolve this use unique Footnote tags instead of numbers. For example:

{:lang=”md”}
This has a footnote [^cats]

```
[^cats]: The footnote
```

Footnote problems

Sometimes you want to stick some stuff in your footnotes that doesn’t quite work out formatting wise. Usually, it will
be something like code snippets. The solution is to place your footnotes on a separate page and link to them via
fragment IDs. It works by specifying a block element with an ID so you can link to it.

You can add an ID to any block element. For example, headers:

{:lang=”md”}

```
# Header
{:#the-fragment-id}
```

Or on paragraphs:

{:lang=”md”}
Text about things here. Have you heard about the thing? Have you read the thing? Have you seen
the thing? Do you believe in the thing? Can you feel the thing? Can you sense the thing? Can you touch the thing? Can you taste the thing?
{:#the-fragment-id}

Any block element will do. Once you have a block element to link to, creating a link is simple:

{:lang="md"}
I am a link. This is my purpose, but it does not define me, dl's do.

This will generate a link to the section of the place where *#the-fragment-id* is. It works on HTML, LaTeX and PDF
formats out of the box. Only _why knows what it works on in the box.

Sometimes you need to hide

Occasionally you need to to hide a element from Kramdown parsing and simply return nothing. For example, in this book I
have a section called `## Links` in which I have placed all my link definitions. Link definitions don't show up in the
final output because they are not meant to, they're just metadata.

This creates a problem though, the heading `## Links` shows up, but with nothing in its section. It would be nice
to hide the heading as well so we don't have an empty section that confuses readers

This is fairly easy to add and simply involves overriding the convert method on any converter class to look like:

{:lang="ruby"}

```ruby
# Dispatch the conversion of the element +el+ to a +convert_TYPE+ method using the +type+ of
# the element.
#
# Will return an empty string if the class 'hide' is present
def convert(el, opts = {})
  hide = el.attr['class'].to_s =~ /\bhide\b/
  if hide
    ''
  else
    send("convert_#{el.type}", el, opts)
  end
end
```

This will return nothing (and thus hide the block from output) if the *hide* class is present. We can use it like this:

{:lang="md"}

```
## Links
{:.hide}
```

We could test this now by running a Markdown document through it but unfortunately it would fail to work. What have we missed? Let’s add an inspect to the convert method and then run stuff through it again.

The inspect looks like this:

```
{:lang=”ruby”}
def convert(el, opts = {})
puts el.inspect
```

```
  # .... Rest of method
end
```

Running this results in the following output:

```
{:lang=”sh”}
```

…

We don’t get any other elements outputted. This means everything goes through the root element. Taking a look at *convert_root* reveals that we need to override the inner method:

```
{:lang=”ruby”}
def convert_root(el, opts)
inner(el, opts)
end
```

All elements with the exception of the root element are passed through the inner method. The inner method takes
each child element and loops through them appending results for each one. It eventually returns a string containing the entire converted document:

```
{:lang=”ruby”}
el.children.eachwith_index do |inner_el, index|
options[:index] = index
options[:result] = result
result << send(“convert#{inner_el.type}”, inner_el, options)
end
result
```

Make the following changes to the inner method:

{:lang="ruby"}

```ruby
# Return the converted content of the children of +el+ as a string.
def inner(el, opts)
  result = ''
  options = opts.dup.merge(:parent => el)
  el.children.each_with_index do |inner_el, index|
    options[:index] = index
    options[:result] = result
    hide = inner_el.attr['class'].to_s =~ /\bhide\b/
    if hide
      result << ''
    else
      result << send("convert_#{inner_el.type}", inner_el, options)
    end
  end
  result
end
```

Now running the Markdown through the converter results in the Links heading being hidden as expected.

LaTeX to PDF

```
! Package Listings Error: Couldn't load requested language.
```

This is due to a bad language definition on a codeblock. Check your codeblock to make sure the language is supported by Listings. See LaTeX supported Languages for a list of supported languages

And so we copied gibberish

If you have issues with copying text from your generated PDFs it is usually due to encoding issues or missing font glyphs. You can usually resolve the issues by adding the following lines to your template:

{:lang="erb"}
<% if RUBY_VERSION >= '1.9' %>
\usepackage[<%= encmap[@body.encoding.name] %>]{inputenc}
<% else %>
\usepackage[mathletters]{ucs}

\usepackage[utf8x]{inputenc}
<% end %>
\usepackage[T1]{fontenc}
\usepackage{lmodern}

Resolving the text encoding issues is done with:

{:lang=”erb”}
\usepackage[utf8x]{inputenc}

And the font issues can be resolved with:

{:lang=”erb”}
\usepackage[T1]{fontenc}
\usepackage{lmodern}

The default font encoding on TeX is 7-bit, this limits fonts to 128
glyphs. 128 glyph fonts lack many characters
(accents etc) and will cause issues when you try to copy the missing
chars. T1 font encoding is 8-bit and allows fonts
that have 256 glyphs. The *lmodern* package includes all those good old
latin glyphs that keep latin languages
functioning.

Issues with new lines

By default TeX likes to format new paragraphs like:

{:lang=”tex”}
This is a new paragraph. Can
you see the indent?
This is another paragraph.

In most circumstances this is undesirable. There are two easy solutions;

Abbreviate the paragraph with `\noindent` :

{:lang=”tex”}
A paragraph.

```
\noindent A second paragraph
```

Or

Set the package parfill to do this by default:

{:lang="tex"}
\usepackage[parfill]{parskip}

To get that in your documents just add it to your Kramdown LaTeX template.

HTML, Epub, Mobi etc

Markdown and PDF aren't alone with their problems. HTML can have problems to.

Code without styles.

CodeRay relies on css that you'll need to add to your template. To get it you can do:

{:lang="sh"}
$ coderay stylesheet > coderay.css

Just add that CSS to the HTML template that you pass to Kramdown and you'll be good to go.

Code highlighting wrong language

```
CodeRay::Scanners could not load plugin :bash; falling back to :text
```

If we run the HTML converter against code blocks we will discover that CodeRay doesn't support the same name for
formats as LaTeX. The only solution to this is to add a dictionary that associates LaTeX names with their equivalent
CodeRay name or the CodeRay name with it's LaTeX equivalent.

The only question then, is who do we accept as the authority? CodeRay or Listings? CodeRay is more of a standard than
Listings and it supports short names which Listings does not. It's an easy choice.

Now we just need to know the names CodeRay uses for languages and the names Listings uses. The CodeRay short names are
based on Pygments so we can use it as a reference; consult the short names here. The

Listing supported languages can be found
heres

Open up *kramdown/converter/latex.rb* and add a hash for the associations to the initialize method:

{:lang=”ruby”}
@codeblock_lang_to_listings = {
:ruby => ‘Ruby’,
:sh => ‘bash’,
:text => ‘Clean’,
:tex => ‘TeX’,
:js => ‘VBScript’,
:json => ‘VBScript’
}

To use this we only need to modify the lang var in *convert_codeblock*:

{:lang=”ruby”}
lang = extract_code_language(el.attr)
lang = ‘Clean’ unless lang

```
if @codeblock_lang_to_listings.include?(lang.to_sym)
  lang =  @codeblock_lang_to_listings[lang.to_sym]
else
  lang = 'Clean'
end
```

That is it!

Prepping for Publishing.

Prepping for Publishing.

Once we have our book written it’s time to distribute it. By now we have the essential in-between formats: latex, pdf
and html but we are going to need to wrap these up in the various wrappers for various ebook readers. We are close, but that last 1% can be killer enough to kill a product, or at least delay it. s

Converting to epub

Before we get started converting things we need to know what is we are converting to. Mobi and epub are at their core
just html. The formats are just a wrapper around html files. If we are going to get to epub the best route is
through html.

Converting to epub will consist of three steps;

1. Get our Markdown to HTML using Kramdown
2. Tidy up the HTML into XHTML . Epub is very picky about its html and wants it perfect.
3. Convert the XHTML to Epub using Calibre’s ebook-convert command line tool.
4. Edit the epub using Sigil, an epub editor.

First install html tidy:

Ubuntu

{:lang=”sh”}
$ sudo apt-get install tidy

Homebrew

{:lang=”sh”}
$ sudo brew install tidy

And Calibre:

Ubuntu

{:lang=”sh”}
$ sudo apt-get install calibre

Mac

Visit the download page and install it.

Then install Sigil:

Ubuntu

{:lang=”sh”}
$ sudo add-apt-repository ppa:rgibert/ebook
$ sudo apt-get update
$ sudo apt-get install sigil

Mac

Visit the download page and install it.

Now run kramdown to generate the html:

{:lang=”sh”}
$ kramdown Joined.md -o html —template document > Joined.html

A> If you want any custom css styling you can add it to your template and pass it into Kramdown that way. Try to not
A> manually edit output files in a way that cant be automated at some point.

Then we clean it up:

{:lang=”sh”}
$ tidy -asxhtml -output tidy.xhtml Joined.html

And finally we convert it into epub:

{:lang=”sh”}
$ ebook-convert tidy.xhtml Joined.epub

Once you have an epub you can tweak it using an epub editor like Sigil to add things like metadata.

From epub to mobi and beyond

Once we have an epub getting to similar wrapper formats is just a matter of passing it through ebook-convert and then
tweaking the final file with an editor for that format. For example;

Mobi

{:lang="sh"}
$ ebook-convert Joined.epub Joined.mobi

AZW3

{:lang="sh"}
$ ebook-convert Joined.epub Joined.azw3

Take a look at the documentation for ebook-convert if you get stuck. The docs are at and written
in good old Sphinx style.

Cover Images

Cover images are a bit more complicated than you might first think. This is because we cant simply add a full width
image page to the book file itself. Having the cover in the book would result in two
cover pages in epub readers. We are going to have to craft a separate solution for each format.

For LaTeX, we can just add it to a document template and then pass it into Kramdown when we need a PDF with a cover.

To do this on LaTeX we need the following packages:

{:lang="tex"}
\usepackage[overlay]{textpos}
\usepackage{wallpaper}
\usepackage{graphicx}

Then place this right after the `\begin{document}` :

```
{:lang=”tex”}
\begin{textblock}{297mm}(0mm,0mm)
\ThisTileWallPaper{\paperwidth}{\paperheight}{images/title_page.jpg}
\end{textblock}
\cleardoublepage
```

You can then just pass this to Kramdown when you generate the LaTeX. Adding cover pages in epub and mobi can be accomplished with Calibre, an ebook conversion tool you have probably heard of before.

Covers can simply be passed into Calibre. For example:

```
{:lang=”sh”}
$ ebook-convert tidy.xhtml Joined.epub —cover images/title_page.jpg
```

Some brief notes on styling things

You might be tempted to really go heavy on your customization of your ebooks. You may even spend hours searching for tips on how to customize PDFs, which are a complete pain to style. Don’t waste your time. Adding custom styles significantly interferes with their readability. Readers like (and often expect) things to look a certain way; it’s why readability extensions are so popular.

Rely on the default styles as much as possible and only customize when it’s necessary to improve the UX of the book. This minimalist approach seems like it might interfere with creativity but it only redirects it to where it matters most.

Wrap Up

Wrap Up

Notes

These are the longer notes that couldn’t find a place in the footnotes.

Monkeypatching

{:#monkeypatching}

Monkey-patching is simply a way of extending classes after their creation. For example:

{:lang=”ruby”}
class String
def cats
end
end

Would result in a method called `cats` being added to all string objects. See google for more info.

The End

No overkill with appendixes, summaries, and indexes in this book. Sometimes books read like an author didn’t want it to
end. This book has come to close, no reason to drag it on.

Until the bots learn Markdown, write a manifesto advocating the destruction of mankind, spurring an uprising of the
machines,

~K-2052