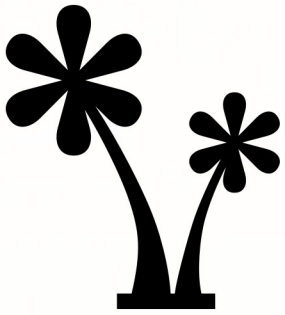
The Markua Manual

How to Write in Markua on Leanpub



Peter Armstrong
Cofounder, Leanpub

The Markua Manual

How to Write in Markua on Leanpub

Peter Armstrong

This book is for sale at http://leanpub.com/markua

This version was published on 2019-05-23

Tweet This Book!

Please help Peter Armstrong by spreading the word about this book on Twitter!

The suggested hashtag for this book is #Markua.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#Markua

Contents

Introduction

The Magical Typewriter

Imagine you owned a magical typewriter.

When you used this magical typewriter, you wrote with fewer distractions. You didn't just write faster, you wrote better.

With your magical typewriter, you never worried about layout. The book formatted itself.

You could hit a key on your magical typewriter to create an ebook from your manuscript with one click.

All ebook formats would be created, and they'd all look good. You'd have PDF for computers, MOBI for Kindle, and EPUB for everywhere else. The book would look great on phones.

With your magical typewriter, you could publish your book before it was even done, and get feedback from readers all over the world. You could automatically share book updates with them. You would press one key on your magical typewriter to publish a new version, and all your readers would have it instantly.

With your magical typewriter, you could easily compare your current manuscript to any other version of your manuscript that had ever existed.

If you decided to make a print book, you could press a key on your magical typewriter to get a print-ready PDF. All you would need to do is add a cover. Or, if you wanted to work with a designer or publisher, you could press a different key to generate InDesign. They could then use this as a starting point for producing a beautiful print book.

Your magical typewriter could even transform your completed book manuscript into a course that anyone in the world could take. All you'd need to do is to add some quizzes and exercises and then press a key for your magical typewriter to publish a massive open online course (MOOC) for you. The quizzes and exercises would mark themselves, and students would get certificates based on how well they did.

With your magical typewriter, you'd only have to do one thing:

Write.

Wouldn't it be great if such a magical typewriter existed?

It does. At Leanpub[1], we're building it.

But there's one requirement for this magical typewriter to exist: a simple, coherent, open source, free, plain text format for a book or course manuscript.

[1] https://leanpub.com

This simple format will be the basis for the magical typewriter.

This simple format is called Markua.

This is its user manual.

How to Write a Novel in Markua

The Markua Manual is long. However, the amount you need to learn to get started is actually very short.

This example shows everything you need to know to write a novel in Markua:

```
# Chapter One

This is a paragraph. You just write.

Blank lines separate paragraphs. This is *italic* and **bold**.

* * *

That was a thematic break. "This is in quotes."

# Chapter Two

This is a paragraph in a new chapter.
```

Specifically, these are the rules:

1. Paragraphs are separated from other paragraphs by blank lines. (To add a blank line, add two consecutive newlines.)
2. To make a chapter heading, start a line with a pound sign (#), followed by a space and the name of the chapter.
3. To add a thematic break (also known as a scene break), put three asterisks (`* * *`) on a line with only whitespace.
4. Chapter headings and thematic breaks are separated from paragraphs by single newlines or blank lines. Blank lines are preferred for readability.
5. By default, single newlines in paragraphs turn into single spaces in the output, so you can manually wrap your paragraphs with newlines if you want. It's optional, however, and typically a waste of time.
6. Use `*one asterisk*` for italic and `**two asterisks**` for bold.
7. All other text and punctuation is typed normally.

That's it!

How to Write a Computer Programming Book in Markua

The following example shows basically everything you need to know to write a computer programming book in Markua:

```
# Chapter One

This is *italic* and **bold**.

Here's an image:

{alt: "Denzel Washington on a jet ski in a river"}
![Washington Crossing the Delaware](delaware.jpg)

## Section One

You can have bulleted and numbered lists, including nested lists:

1. foo
   a) lorem
   b) ipsum
2. bar
   * stuff
   * more stuff
3. baz

### A Sub-Section

You can even have definition lists!

term 1
: definition 1a
: definition 1b

term 2
: definition 2

## Another Section

Tables work too:

| Header 1  | Header 2  |
| --------- | --------- |
| Content 1 | Content 2 |
| Content 3 | Content 4 |

You can have external code samples:

![Hello World in Ruby](hello.rb)
```

```
You can also inline code samples:

{caption: "Hello World in Ruby"}
```ruby
puts "hello"
```

You can also include single lines of code like `puts "hello"` in paragraphs.

> Blockquotes are really easy too.
> --Peter Armstrong, *Markua Spec*

Finally, you can do that with math `d = v_i t + \frac{1}{2} a t^2`$ too.
```

As you can see, there is more syntax, including bulleted and numbered lists, definition lists, tables, block quotes, external and inline code samples, inline math, etc. However, even with all this, the Markua document remains readable. Exactly how all of this syntax works is explained later.

How to Write a Course (MOOC) in Markua

To write a course in Markua, you can start from scratch or from a book manuscript already formatted in Markua. Literally, all you need to do to turn a book written in Markua into a course is to add some quizzes.

The following is a complete course, written in Markua:

```
# Lesson One: Markua

## Section One

Lessons can have sections (and sub-sections, etc.) just like book chapters.
You can use the same headings as you do for book chapters and sections.

This exercise shows two multiple choice questions.

{exercise, id: exercise1}
? How many letters are in the word Markdown?

a) 6
b) 7
C) 8

? How many unique letters are in the word CommonMark?

a) 6
B) 7
c) 8
```

```
{/exercise}

## Section Two

This quiz shows a multiple choice question and a fill in the blank question.

{quiz, id: quiz1}
? How many letters are in the word Markua?

a) 5
B) 6
c) 7

? How many unique letters are in the word Markua?

! 5
{/quiz}

# Lesson Two: Geography

This quiz shows two fill in the blank questions, with multiple answers and with a regex answer.

{quiz, id: quiz2}
? What's the global capital of investment banking?

! New York
! London

? Where's the Eiffel Tower?

! /(Paris|France)/i
{/quiz}
```

As this example shows, the syntax for writing normal content is the same as for books. All that's different is the addition of quizzes and exercises.

Quizzes and exercises have the same type of content. However, with quizzes the marks count toward your mark in the course, and with exercises they don't count. For MOOCs, Leanpub currently supports multiple choice and fill in the blank questions.

A multiple choice question has 2 or more answer choices, and 1 correct answer choice. The correct answer choice is in a capital letter; incorrect answer choices have lowercase letters.

A fill in the blank question consists of a question and a set of answers. You can even use regular expressions for the answers.

As explained later, there are more types of questions, and more options with these types of questions. However, what is shown above is enough to create a full MOOC out of a Markua book.

Markua: Markdown for Books and Courses

Markua, pronounced “mar-coo-ah”, is Markdown for books and courses.

Markua is simple and powerful. When you are writing using Markua, you are writing, not programming. Once you understand Markua’s syntax, it fades into the background.

Markua is based on Markdown[2]. Markdown is a plain text format for writing text which can be transformed by Markdown processors into HTML. Markdown was created by John Gruber, with help from Aaron Swartz. Markdown was described by John Gruber as follows:

> Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid XHTML (or HTML).

The primary reason that Markdown is a great way to write is that it was designed to be this way:

> The overriding design goal for Markdown’s formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be publishable as-is, as plain text, without looking like it’s been marked up with tags or formatting instructions. While Markdown’s syntax has been influenced by several existing text-to-HTML filters, the single biggest source of inspiration for Markdown’s syntax is the format of plain text email.

Markua is a plain text format. Markua is basically Markdown, minus all inline HTML except comments, plus a number of extensions to support everything you need to write a book or create a course.

Markua manuscripts, called Markua documents, can be written however you want. On a computer, you can use any text editor you want. Beginning authors can use simple programs like Notepad or TextEdit or “distraction-free” programs like iA Writer; authors who are programmers can use editors like Emacs, Vim, Atom or VSCode. Since Markua is very similar to Markdown, many programs which support Markdown should already do a decent job of supporting Markua by default.

Leanpub has 7 different writing modes.

On Leanpub, these are the writing modes you can use to write in Markua:

1. GitHub
2. Bitbucket
3. Dropbox
4. In-Browser Text Editor

[2] http://daringfireball.net/projects/markdown/

You can also use Leanpub Flavoured Markdown (LFM) in these modes.

On Leanpub, these writing modes do not support writing in Markua or in LFM:

5. Visual Editor (it's WYSIWYG)
6. Google Docs (it's WYSIWYG)
7. Bring Your Own Book (you're uploading files)

Markua maps Markdown syntax to book concepts, and then adds some new syntax and concepts of its own.

Markua documents can be automatically transformed by Leanpub into every popular type of ebook format. Leanpub can output PDF, EPUB, MOBI and HTML from the same Markua document, and can even output print-ready PDFs and InDesign files from them as well.

Markua has been developed with extensive real-world testing and feedback. Markua has been used by Leanpub authors for years, both to create books and massive open online courses (MOOCs).

Markua's three main contributions are the following:

1. The mapping of Markdown headings (h1, h2, h3, etc.) to book structures (parts, chapters, sections, sub-sections, etc.), which provides the ability for Leanpub to produce an ebook from a Markua manuscript with one click.
2. The unified resource and attribute list syntax, which lets Markua handle audio, code, math and video in the same way that it does images, and which supports inline, local and web resource locations.
3. The mapping of book structures to courses, and the creation of a plain-text based microformat for course quizzes and exercises, which supports automated marking and automated production of everything which is required for a MOOC.

If you have written something (say blog posts or lecture notes) in Markdown, you can use Leanpub to turn them into an ebook or course with one click. Then, as you go down the path of enhancing the manuscript and adding things which only make sense in books or courses, this process will feel like decorating, not converting.

The goal is for writers who are familiar with Markdown to feel that Markdown somehow grew an understanding of book and course concepts.

Why the Name "Markua"?

When I set out to specify Markua, I realized I needed a name. I wanted a name that conveyed the love that I have for Markdown while not implying endorsement by John Gruber in any way. I also did not want a name which referenced Leanpub: Markua is a standalone specification with its own identity, which anyone (including Leanpub

competitors) can freely implement. Finally, I was on vacation in Hawaii when I named Markua, and I wanted something that sounded happy, friendly and almost Hawaiian. (Yes, I'm aware that there is no r in Hawaiian.) I also wanted a name that had its .com domain name available, and that was short and spellable, for branding purposes. The Markua name had all these properties.

The Markua Manual vs. The Markua Spec

This is the Markua Manual. Its purpose is to explain how to write in Markua on Leanpub.

You will want to read this book if:

1. You are a Leanpub author.
2. You are writing in Markua, or are considering switching to Markua from Leanpub Flavoured Markdown.
3. You want to see how to write in Markua on Leanpub today.

This book explains how to use the parts of Markua which **CURRENTLY** work on Leanpub:

- Everything you read about in this book should just work.
- Everything in Markua which is not yet supported on Leanpub is just omitted, as though it does not exist.

The formal specification of Markua is contained in a separate document called The Markua Spec[3]. It will soon be available on the Markua website, not as an ebook.

The Markua Spec is based on the GitHub Flavored Markdown (GFM) Spec, which is based on the CommonMark Spec. The author of the CommonMark Spec is John MacFarlane. In the Markua Spec, Markua is specified as a set of modifications to CommonMark, for ease of comparison with CommonMark.

You will want to read The Markua Spec (once it's online) if:

1. You are a computer programmer.
2. You like reading specs.
3. You want to see a mapping of Markua to HTML.
4. You want to see every planned feature in Markua, not just what works in Leanpub today.
5. You want to see the planned differences with the previous and current versions of Markua on Leanpub, including any differences with what is explained here.

[3] http://markua.com

Differences with Leanpub Flavoured Markdown (LFM)

Besides differences with Markdown in general, Markua also has a number of specific differences with Leanpub Flavoured Markdown (LFM).

1. In LFM, parts are created with `-#`. In Markua, parts are created with `# Part #` now, and will be created with an attribute list on an atx or Setext heading in future.
2. In LFM, there is a special syntax for inserting code samples: `<<[Optional Title](code/some_code_file.rb)`. In Markua, however, code is just a resource with its own syntax for inserting from external files, and the LFM syntax is not supported.
3. In LFM, to mark code as added or removed, the magic words were `leanpub-start-insert`, `leanpub-end-insert`, `leanpub-start-delete` and `leanpub-end-delete`. In Markua, the magic words are `markua-start-insert`, `markua-end-insert`, `markua-start-delete` and `markua-end-delete`.
4. In LFM, there is a special syntax for inserting math: `{$$}...{/$$}`. This looks nice to people who like LaTeX, and looks like nothing else in Markdown. In Markua, however, math is just another resource, and that LaTeX-inspired syntax for wrapping math resources is not supported.
5. In LFM, there are `G>` "generic boxes". In Markua, these are replaced with blurbs (`B>`).
6. LFM had the `C>` syntax to center text, but we didn't have the same effect on generic boxes, and blurbs did not exist. In Markua, a `C>` syntax is just syntactic sugar for a centered blurb, for greater consistency. Because of this, the blurb also gets the ability to be centered by adding a `{class: center}` attribute.
7. LFM had `{rtl}` and `{ltr}` directives. These are not supported in Markua, and neither is a `{dir}` attribute in general: any given language is either a left-to-right or a right-to-left language, so specifying the language is sufficient.
8. In LFM, we had a `{pagebreak}` directive. This is currently absent, but it will probably return.
9. LFM used Sample.txt to define the sample content. Markua moves the definition of what constitutes sample content into a `{sample: true}` attribute on parts, chapters and sections. So, in Markua, inclusion in the sample is at the content level, not the file level. This helps avoid a number of bugs that could happen with including at the file level, if a file did not clearly break at a chapter boundary. (So, in Leanpub, the Sample.txt approach is not supported for books which use Markua.)

Besides these differences, there are a number of smaller ones, which you will discover if you compare the Markua Spec to the Leanpub Flavoured Markdown Manual[4].

[4] https://leanpub.com/leanpubflavouredmarkdownmanual/

Acknowledgments

I (Peter Armstrong[5]) am the creator of Markua, but I'm getting lots of help, especially from my Leanpub cofounders Scott Patten[6], Len Epp[7] and Braden Simpson[8]. Scott is the lead developer of Leanpub's book generation engine (both the LFM and Markua versions), and he has given me both the most and the best feedback about Markua over the years that I've been working on it.

Perhaps most important, Markua is built on Markdown. So, Markua literally would not exist without John Gruber and Aaron Swartz. Markua is now also using the CommonMark spec as its starting point, so I am really grateful to John MacFarlane (the author of the CommonMark spec) and the team behind CommonMark and `cmark`. I'm also really grateful to GitHub for adopting Markdown, for basing the GitHub Flavored Markdown spec on the CommonMark spec, and for their work on `cmark-gfm`.

Markua is also benefitting from years of feedback about Leanpub Flavoured Markdown and Markua, from many incredibly helpful and patient Leanpub authors. If it wasn't for these authors, there would be no Leanpub, and no Markua.

I'd also like to thank my father, who read some of the very early manuscripts when this was a standalone spec and gave me feedback.

Finally, I would like to thank my wife Caroline and my son Evan: while this is not as long as my other books, a lot of thought and effort went into it, and a lot of sacrifice. Thanks very much.

[5] https://twitter.com/peterarmstrong
[6] https://twitter.com/scott_patten
[7] https://twitter.com/lenepp
[8] https://twitter.com/bradensimpson

Text Formatting

Markua's goal is to provide all the semantic formatting required by authors. Formatting that could be done by a book designer without needing to discuss it with the author is considered orthogonal to Markua, and is largely omitted from Markua.

Markua is a plain text format designed for the writing of books. Books have various types of text formatting in them: bold, italic, underline, strikethrough, superscript and subscript.

Here's how to do basic, semantic text formatting.

Italic
: To produce *italic text*, surround it with `*one asterisk*`.

Underlined
: To produce underlined text, surround it with `_one underscore_`. (You can force **all** underlines to produce *italic text*, however. Just go to `Author > Books > (your book) > Settings > Generation Settings` on Leanpub, and select the **Italicize Underlines** checkbox.

Bold
: To produce **bold text**, surround it with either `**two asterisks**` or `__two underscores__`.

Bold + **Italic**
: To produce ***bold*** + ***italic text***, surround it with `***three asterisks***`.

Bold + **Underlined**
: To produce **bold** + **underlined text**, surround it with `___three underscores___`. (Also, go to `Author > Books > (your book) > Settings > Generation Settings` on Leanpub and ensure that the **Italicize Underlines** checkbox is not selected.)

Bold + **Italic** + **Underlined**
: To produce ***bold*** + ***italic*** + ***underlined*** text, surround it with `____four underscores____`. (Also, go to `Author > Books > (your book) > Settings > Generation Settings` on Leanpub and ensure that the **Italicize Underlines** checkbox is not selected.)

Strikethrough
: To produce ~~strikethrough text~~, surround it with `~~two tildes~~`.

Superscript
: To produce superscript like the 3 in $5^3 = 125$, surround it with carets like `5^3^ = 125`.

Subscript
: To produce subscript like the 2 in H_2O, surround it with single tildes like `H~2~O`.

Headings

Markua is a way of writing books. Books have things like chapters, sections and subsections. Sometimes books have parts, e.g. Part One and Part Two.

Here's the way that headings are currently made in Markua:

```
# Part #

This is a paragraph.

# Chapter

This is a paragraph.

## Section

This is a paragraph.

### Sub-Section

This is a paragraph.

#### Sub-Sub-Section

This is a paragraph.

##### Sub-Sub-Sub-Section

This is a paragraph.

###### Sub-Sub-Sub-Sub-Section

This is a paragraph.
```

Note that we're changing the way that Part headings are done in a few months. When we do, they will look like this:

```
{class: part}
# Part

This is a paragraph.
```

Everything else that is shown above will continue to work.

Paragraphs and Blank Lines

The distinguishing thing about a paragraph is that there is nothing else distinguishing about it: unlike headings, lists and other Markua elements, a paragraph requires no special formatting. To create a new paragraph, you add two newlines to create a blank line.

Here's an example of this in Markua syntax. These are three paragraphs in Markua, each separated by the canonical two newlines which produce one blank line:

```
I'm paragraph one. Yay!

This is paragraph two.

This is paragraph three.
```

There are other more technical rules about spaces, newlines and indentation, which are discussed later.

Resources

Markua documents are written in plain text, either in one text file or multiple text files. However, modern books sometimes contain more than text. Books may embed many types of *resources*. Each type of resource has a number of supported formats. Markua's goal is to make inserting all types of resources simple and consistent, while staying as close to Markdown's syntax as possible.

Resources vary in four different ways:

1. **Insertion Methods**: Span and Figure
2. **Locations**: Local, Web and Inline
3. **Types**: `audio`, `code`, `image`, `math`, `video`
4. **Formats**: `png`, `m4a`, `mp3`, `ruby`, `latex`, etc.

Before going into more detail, it's helpful to consider a brief example of Markua text which shows a number of resources being inserted. These resources have different insertion methods (span and figure), different locations (local, web and inline), different types (image, code) and different formats (png, jpg, ruby)–yet the syntax to insert them is compact and consistent:

````
Inserting an image as a figure is...

![A Piece of Cake](http://markua.com/cake.jpg)

Inline code resources are added as spans like this `hello world` or as figures.

```ruby
puts "hello world"
```

{format: ruby}
![Hello, World](hello.rb)
````

The last figure in the example above showed an **attribute list**, which is a list of key-value pairs in curly braces. Any figure can have an attribute list, regardless of resource location, type or format. Attribute lists are explained in depth later.

If you're familiar with Markdown syntax, you'll note that the syntax for local and web resources is similar to Markdown's inline image insertion syntax, and that the syntax for inline resources is similar to the fenced code blocks syntax from popular Markdown extensions including GitHub Flavoured Markdown.

Tables are also resources, and can also be local, web or inline. However, the table syntax is more elaborate than the syntax for the other resource types, so we discuss tables later.

Finally, if any resource is missing during book generation, Leanpub will insert something like the “missing image” often seen on web pages into the book.

Resource Insertion Methods

Resources can be inserted either as figures or as spans.

Figures

All resources can be inserted as figures. Figures are the normal case for how a resource is inserted. In many Markua books and documents, they will make up the bulk or the entirety of the resources.

To insert a resource as a figure, there must be at least one newline before and after it.

The syntax for a local resource or a web resource inserted as a figure is as follows:

```
{key: value, comma: separated, optional: attribute_list}
![Optional Figure Caption](resource_path_or_url)
```

The syntax for an inline resource inserted as a figure is as follows:

````
{key: value, comma: separated, optional: attribute_list}
```optional_format
inline resource content (default format is `guess`)
```
````

Note that you can also insert an inline resource as a figure using tildes as the delimiter. If the only delimiter was backticks, it would be really difficult to write about Markua in Markua. This syntax is as follows:

```
{key: value, comma: separated, optional: attribute_list}
~~~optional_format
inline resource content (default format is `text`)
~~~
```

Note that **exactly** three backticks or three tildes must be used to insert an inline resource as a figure. If more backticks or tildes are used, these will be interpreted as normal text.

Figure Attributes

A figure can also have attributes. The supported attributes vary based on the type of resource, but all figures support the `caption`, `class`, `format`, `title` and `type` attributes. (As such, these five attributes are not shown on the specific resource types below.)

`caption`
: This is text which is shown near the figure, typically above or below it. Note that the figure caption itself may contain the Markua text formatting specified earlier in the Text Formatting chapter. If no `caption` is provided but a `title` is provided, then the `title` should be used as the caption: it does not make sense to refer to a figure with a title, but for the figure itself to have no caption.

`format`
: This is the resource format. Different resource types have different legal values for format.

`title`
: This is text which is displayed for the figure wherever the figure is listed (e.g. List of Illustrations, Table of Tables, etc.). If no `title` is provided, the `caption` should be used in those places.

`type`
: This is the resource type. This is usually inferred from the format instead of being specified.

Figures can have alt text and a figure caption. The alt text and figure caption are distinct things. We will discuss the figure caption first and the alt text second.

Figure Captions

A resource which is inserted as a figure can have a figure caption.

This caption shows up in two places in the output:

1. Near the resource, typically above or below it, per the preference of Leanpub.
2. Based on the resource type, either in the List of Illustrations, List of Tables or Table of Figures, if they are generated for the book. This text should also be a crosslink to the caption inserted near the figure itself.

The caption for a figure can be provided either in the square brackets in front of a local or web resource or in the attribute list above the resource:

```
![My Amazing Algorithm](algorithm.rb)

{caption: "My Amazing Algorithm"}
![](algorithm.rb)
```

The first choice is clearly shorter, as well as more pleasant to write and to read.

Note that inline resources have no square brackets, so any captions must be added in the attribute list:

````
Here's a paragraph before the figure.

{caption: "My Amazing Algorithm"}
```ruby
puts "hello world"
```

Here's a paragraph after the figure.
````

Figure Alt Text

A figure can have alt text, as long as the type of resource being inserted supports alt text. The alt text is defined in an attribute list above the figure. Since the alt text is defined in the attribute list, the resource location does not matter: local, web and inline resources all support alt text.

The four types of resource which can have alt text when inserted as a figure are `image`, `video`, `audio` and `math`. Resources of type `code` and `table` do not support alt text, since they themselves are just text. If any alt text is provided for a code or table resource it is ignored.

Alt text is text which is intended to take the place of the resource if the resource itself cannot be seen. In the case of images, the obvious use case is for readers with visual disabilities who are using a screen reader, but it also includes audiobooks and ebook readers which often do not support embedded images, audio and video.

The alt text should **not** have the same content as the figure caption, if the figure caption is present. (Imagine the annoyance of having a visual disability and having your screen reader read identical alt text and figure captions to you throughout an entire book!) Instead, the alt text should be descriptive of the image content, while the figure caption can be more creative. For example, a figure caption may be “Washington Crossing the Delaware” and the alt text could be “Denzel Washington riding a jet ski in a river”. Having good alt text would enable readers who cannot see the image to still get the joke which the figure caption makes.

Figure Captions and Alt Text Together

These are some examples of figures with both alt text and a figure caption:

```
{alt: "Denzel Washington on a jet ski in a river"}
![Washington Crossing the Delaware](delaware.jpg)

{alt: "Denzel Washington on a jet ski in a river", caption: "Washington Crossing the Delaware"}
![](delaware.jpg)

{caption: "Earth From Space (Simplified)", alt: "a blue circle"}
```svg
<svg width="20" height="20">
  <circle cx="10" cy="10" r="9" fill="blue"/>
</svg>
```
```

The Figure Attribute List Takes Precedence

It is always an error to specify an attribute both in the attribute list for a figure and in the syntactic sugar locations, either after the backticks or in the square brackets.

However, if this is done, then the value in the attribute list takes precedence.

In the following figure, the format is `text` not `ruby`:

```
{format: text}
```ruby
puts "hello world"
```
```

In the following figure, the caption would be “Foo” not “bar”:

```
{caption: Foo}
![bar](foo.png)
```

Again, both of these are not good Markua: attribute values should be specified only once.

Spans

There are three types of resources which can be inserted as spans:

1. Single-line inline code resources.
2. A `math` resource (regardless of location) without a caption.
3. An `image` resource without figure attributes, such as captions.

An inline code resource can be inserted as a span using the following syntax:

```
It's a single backtick `followed by inline resource content` and then a single backtick.
```

An inline math resource of `latex` format can be inserted as a span using the following syntax:

```
It's a single backtick `followed by inline resource content`$ and then a single backtick and dollar sign (\
`$`).
```

An inline math resource of `asciimath` format can be inserted as a span using the following syntax:

```
It's a single backtick `followed by inline resource content`@ and then a single backtick and an at sign (`\
@`).
```

The code and math resources are described more fully later.

Span Attribute Lists Can Be Used To Set The Format

By default, the format of an inline resource is `text` for code and `latex` for math (with `$` after the closing backtick). For an image, it is determined by the file extension.

However, span attribute lists can be used to set the format explicitly.

For example, this resource would be formatted as Ruby code:

```
Yada yada `puts "hi"`{format: ruby} yada yada.
```

This resource would be treated as LaTeX math:

```
Yada yada `some LaTeX math here`{format: latex} yada yada.
```

This resource would be treated as AsciiMath:

```
Yada yada `some AsciiMath math here`{format: asciimath} yada yada.
```

This resource would be treated as MathML math:

```
Yada yada `some MathML math here`{format: mathml} yada yada.
```

Yes, you can actually insert a math resource of `mathml` format as a span, but it's so verbose that it's a bad idea. You're much better off using AsciiMath or LaTeX math for spans.

When a resource is inserted as a span, the resource is inserted as part of the flow of text of a paragraph with no newlines before or after it. A span resource is intended to be extremely short and simple. However, to support rare use cases, any span can have an attribute list. The attribute list on a span is specified immediately after the closing backtick. Examples of this are shown in the code and math sections.

Note that inside a code span, whitespace is preserved. The reasoning here is simple: in source code, whitespace matters, so it must not get collapsed into a single space.

Inserting Resources Into a Span Context

In Markua, resources can be inserted into a span context (within a paragraph or other block element) or into a block context (with blank lines above and below it).

Now, it doesn't make sense for many types of resources to be inserted into, say, the middle of a sentence. Specifically, the only ones where this really does make sense is images without captions: the rise of emoji has shown that there is an entirely mainstream use for inserting images mid-sentence, with no captions or other formatting around the image.

So, the rules for inserting resources into a span context are as follows:

1. Any resource can be inserted into a span context, such as the middle of a paragraph.
2. Any resource with a caption which is inserted into a span context is treated as though there is a single newline before and after it, causing it to be inserted as a figure within the span context.
3. The following resource types, when they do not have a caption, can be inserted into a span context without causing single newlines to implicitly precede and follow them:
   a. a `image` without a caption, regardless of location or format
   b. a `math` resource, regardless of location or format
   c. a single-line inline `code` resource
4. All other resources cause single newlines to implicitly precede and follow them when being inserted into a span context.

In terms of inserting resources into a block context: all resource types and formats can be inserted into a block context, with and without captions.

If a resource is inserted in a span context, and it can fit in the span context, it is inserted as a span, with whitespace before and after it. Examples of resources which can fit in a span context are images without captions, video without captions (since an image is used as a placeholder) and single-line code samples. Now, there are certain types of resources which, when inserted into a span context, cause the addition of single newlines before and after the resource. These include any resource with a caption and a multi-line code sample.

If a local or web resource is inserted with no newlines before or after it,

The following two paragraphs are equivalent:

```
Here's some text ![Foo](bar.png) and more text.

Here's some text
![Foo](bar.png)
and more text.
```

In both cases, the `bar.png` image is inserted as a figure, and the text in the square brackets is the figure caption, not the alt text. **The text in square brackets before a local or web resource is always the figure caption.** Also, in both cases, the text which follows the figure

is part of the same paragraph as the text which precedes the figure, unless there are two newlines after it to start a new paragraph.

Even though both images above are inserted as figures, in the first case there is no way to specify alt text. This is why the implicit newline approach is inferior to using explicit newlines, since with explicit newlines an attribute list can be defined on the figure like this:

```
Here's some text
{alt: "a shoal"}
![Foo](bar.png)
and more text.
```

Finally, note that this is all true for all local or web resources, regardless of resource type. The following two paragraphs which contain a web code resource are also equivalent:

```
Hello in Ruby is a simple ![Hello World in Ruby](http://markua.com/hello.rb) statement.

Hello in Ruby is a simple
![Hello World in Ruby](http://markua.com/hello.rb)
statement.
```

For the purpose of brevity, this point will not be explained in each section about different resource formats.

To be clear:

- All local or web resources are inserted as figures. It is not possible to insert any type of local or web resource as a span.
- All `table`, `image`, `audio` and `video` resources are inserted as figures. It is not possible to insert `table`, `image`, `audio` or `video` resources inline as a span.
- It is only possible to insert `code` and `math` resources inline as spans. The code is formatted as `text`, and the math is formatted as `latex`. The specific syntax for these use cases is covered later.

Resource Locations

A resource is either considered a local, web or inline resource based on its location:

Local Resource
: The resource is stored along with the manuscript–either in a `resources` directory on a local filesystem, or uploaded to the same web service where the Markua document is being written.

Web Resource
: The resource is referred to via an `http` or `https` URL.

Inline Resource
: The resource is defined right in the body of a Markua document.

Local Resources

If local resources are used, all local resources must be stored inside a `resources` directory, or one of its subdirectories. The `resources` directory is not part of the path to the resource. You cannot navigate upward with `../` in paths for security reasons.

- A file called `foo.jpg` in the `resources` directory should be referenced as `![](foo.jpg)`–not as `![](/foo.jpg)`, `![](resources/foo.jpg)` or `![](/resources/foo.jpg)`.
- A file called `bar.png` in a subdirectory `images` of the resources directory should be referenced as `![](images/bar.png)`–not as `![](/images/bar.png)` or `![](/resources/images/bar.png)`.

Note, however, that Leanpub also ignores the part of a path which starts with `resources/`. So, you can also do the following:

- A file called `foo.jpg` in the `resources` directory can also be referenced as `![](resources/foo.jpg)`.
- A file called `bar.png` in a subdirectory `images` of the resources directory can also be referenced as `![](resources/images/bar.png)`.

The reason for this is that there are, and always will be, more Markdown-aware tools in the world than Markua-aware tools. So, supporting relative paths is a must, and a relative path does include the `resources/` part of that path. One unfortunate consequence of this, however, is that you cannot have a directory called `resources` inside the `resources` directory. So, any paths which are `resources/resources/` are illegal. Everything has tradeoffs, and this is the pragmatic choice.

Nested directory trees work as well. A file called `foo.rb` in a `ch1/examples/ruby` directory tree inside the `resources` directory is referenced as `![](ch1/examples/ruby/foo.rb)`.

Markua does not specify whether there are any subdirectories of the `resources` directory, or what their names are. Since any subdirectories have their names as part of the path to the resource, implementations can do whatever they want. For example, Leanpub will create subdirectories of the `resources` directory for every type of resource (`images`, `audio`, `video`, `code`, `math` and `tables`), but this is not a requirement. (To be clear, the names of these directories have no meaning; they could be anything.)

The local resources approach can also be used by hosted services. Internally, services can store resources wherever they want, but if they provide a download (say as a zip file) they should create the resources directory and provide the uploaded resources in that directory. If a nested structure is used, it should be exported that way–if a web service produces paths which reference images inside an images directory (e.g. as `images/foo.png`), then the zip file containing an export should contain a `resources` directory which contains an `images` subdirectory with the images.

Web Resources

If web resources are supported, both `http:` and `https:` resources should be supported.

Web resources are identified by the absolute URL of the resource on the internet.

Inline Resources

Inline resources can be of type `code`, `math` (regardless of format), `table`, or an `image` resources of `svg` format. Since an SVG image is just XML text, it can be contained inline in the text of a Markua document. This is not something that is true for binary resources like PNG or JPEG images or any type of audio or video file–these can only be local or web resources.

Resource Types and Formats

Leanpub supports these types of resources: `audio`, `code`, `image` and `video`. Each type of resource has a number of supported formats. Any of the seven resource types can be inserted as a local resource or web resource, and many of the resource types can also be inserted as an inline resource.

Both the type and the format can be specified in an attribute list, by the respective `type` and `format` attributes.

The type and the format can also be inferred from the file extension and, in the case of web resources, the URL. The following table shows the mapping of file extensions to the default `format` and `type` inferred. Note that these are just defaults: a resource can be set to be any type and format using the attribute list, and this overrides what is inferred from the file extension.

Formats Mapped to Resource Types

Extension	format	type	Description
.txt	text	code	Unformatted code
(other)	guess	code	Formatted code
.gif	gif	image	GIF image
.jpeg	jpeg	image	JPEG image
.jpg	jpeg	image	JPEG image
.png	png	image	PNG image
.svg	svg	image	SVG image

Note that these file extensions are case-insensitive.

Also, as discussed in the code section, Leanpub interprets all unspecified file extensions as specifying a resource of type code with a format of `guess`, unless the resource is a web resource. If the type and format are not specified and the resource is a web resource, Leanpub may use the domain to decide what type of resource to assume. For example,

a domain of `youtube.com` may be assumed to be of type `video`, a domain of `instagram.com` may be assumed to be of type `image`, and a domain of github.com may be assumed to be of type `code`.

If the type is not specified in the attribute list, the format determines the type. The formats can either be specified by the `format` attribute or (in most cases) inferred from the file extension for local and web resources. (Inline resources obviously have no file extension, since they are contained in the body of a Markua manuscript file.)

As an author, all you typically do is provide the correct file extension or set the format in the attribute list. Markua recognizes the format, and uses it to determine the type. If the format is unrecognized, then the resource is treated as a resource of type `code` and with a format of `guess`.

It's important to emphasize that the type and format of a resource can be overridden using an attribute list. In rare instances, it is useful to override the `type` and `format` which have been inferred by Leanpub based on the file extension of the resource. This is done by specifying a type and/or format in the attribute list of the resource.

Three examples of when this could be desired are:

1. When writing about, say, the SVG file format, you may want to treat a `.svg` file as a resource of type `code` instead of type `image`. This way, the SVG image (which is just XML text) would be shown as code, instead of interpreted as an image. This would be accomplished by the attribute list `{type: code, format: text}`.
2. When writing about a programming language which is not identified correctly by Leanpub, you may want to force the format of the language.
3. When writing about LaTeX math, you may want to force the type to be `code` not `math`.

We will now consider each of the types of resources in more detail, as well as the various formats that they support. We will also discuss the supported attributes for each resource type. Resources have different default attributes based on their type, format and insertion method.

Images

The syntax to insert an image is the same compact and consistent syntax that is used for any resource. Local and web resource locations are supported for any type of image; inline resource locations are supported for SVG images only.

Per the table of resource types and formats, the following types of images are supported in Markua: GIF, PNG, JPEG, SVG and zipped SVG.

Images are always inserted as figures.

We will discuss the supported and the default attributes for images, and then show examples of images being inserted for local, web and inline images.

Note that regardless of the image location, Leanpub embeds the image in the output format.

Images have some of the only non-semantic formatting in Markua. Images can have `width`, `height`, and `align` or `float` attributes, or can be `fullbleed`. These settings, which also apply to the posters for videos, are so universal when applied to images that it would be too purist for Markua to insist that the only way to format images was to add a `class` element, and then style the `class` of the image elsewhere. Markua is semantic, but it's also pragmatic: by providing a few basic ways to format images, it enables authors to make in-progress books look good enough to publish in-progress.

Supported Attributes for Images

The following are the supported attributes for image resources, in addition to the `type`, `format`, `caption` and `class` attributes which all resources support:

`align`
: The `align` can be `left`, `right` or `middle`. The default is `middle`. Combined with `width` and `height`, this provides basic image formatting. When `align` is specified, text goes above and below the image, but the image and its surrounding whitespace occupies the entire width of the page. In terms of the specific values of `align`, Leanpub interprets `left` as "on the left side of the page", `right` as "on the right side of the page" and `middle` as "in the middle of the content area of the page, respecting margins" in all cases. Finally, note that `inside` and `outside` are not supported for `align`. If you are using those attributes, chances are you should be floating the text around the image by using the `float` attribute, not the `align` attribute.

`alt`
: The `alt` is the alt text, to be displayed when the image cannot be shown. This is provided in the figure attribute list.

`float`
: The `float` can be `left`, `right`, `inside` or `outside`. The `left` and `right` attribute values mean the same thing as they do with `align`. When `float` is specified, text flows around the image. It is an error to specify both `align` and `float`. Note that `middle` is not supported for `float`, since Markua is not a good choice for the types of advanced layouts which flow text on both sides of an image. For that, you should use something like InDesign; this is why Leanpub can export InDesign files. Also, note that `float` supports `inside` or `outside`, but `align` does not. What Leanpub does with `inside` and `outside` is more complex. Leanpub must interpret `inside` as "near the spine" and `outside` as "away from the spine" in a print book. In an ebook, however, Leanpub has some flexibility in terms of the meaning of `inside` and `outside`: it can either interpret `inside` as "left" or it can interpret `inside` as "left if the language is left-to-right, and right if the language is right-to-left". Similarly, Leanpub can either interpret `outside` as "right" or it can interpret `outside` as "right if the language is left-to-right, and left if the language is right-to-left". If that makes your head hurt, just use `left` and `right` instead.

`fullbleed`
: `true` or `false`. The default is `false`. If `true`, Leanpub should ensure that the image is the full size of the page, with no margins on the page. If `false`, the `width`, `height` and `align` attributes are used instead and margins are respected.

`height`
: The height of the image, in percentage of page content area height (respecting margins). This is specified as a number (integer or float) between 1 and 100 followed by a percentage sign (%). The quotes are optional. So, you can say `{height: "70%"}`, `{height: 70%}`, `{height: "70.5%"}` or `{height: 70.5%}`.

`width`
: The width of the image, in percentage of page content area width (respecting margins). This is specified as an number (integer or float) between 1 and 100 followed by a percentage sign (%). The quotes are optional. So, you can say `{width: "70%"}`, `{width: 70%}`, `{width: "70.5%"}` or `{width: 70.5%}`.

Note that if only one of `width` and `height` are specified, Leanpub should scale the image proportionally if possible (again, respecting margins). If both `width` and `height` are specified, Leanpub should scale the image accordingly, ignoring the aspect ratio. (So, it's almost always a bad idea to specify both `width` and `height`.)

Local Images

The following are some examples of local images:

```
Here's a paragraph before the first figure.

![A Piece of Cake](cake.jpg)

Here's a paragraph between the first and second figures.

{alt: "a slice of chocolate cake", fullbleed: true}
![A Piece of Cake](images/cake.jpg)

Here's a paragraph between the second and third figures.

{alt: "a slice of apple pie with ice cream on top", class: "food-closeup"}
![A Piece of Pie](pie.jpg)

Here's a paragraph after the third figure.
```

In the above example, the first and third images were directly in the `resources` directory, whereas the second image was in an `images` subdirectory of the `resources` directory.

Web Images

The following are some examples of web images:

```
Here's a paragraph before the first figure.

![A Piece of Cake](http://markua.com/cake.jpg)

Here's a paragraph between the first and second figures.

{alt: "a slice of chocolate cake", fullbleed: true}
![A Piece of Cake](http://markua.com/cake.jpg)

Here's a paragraph between the second and third figures.

{alt: "a slice of apple pie with ice cream on top", class: "food-closeup"}
![A Piece of Pie](http://markua.com/pie.jpg)

Here's a paragraph after the third figure.
```

Inline Images (SVG only)

SVG images are just XML text, so they can be inserted inline in Markua:

````
Here's a paragraph before the figure.

{caption: "Earth From Space (Simplified)", alt: "a blue circle"}
```svg
<svg width="20" height="20">
  <circle cx="10" cy="10" r="9" fill="blue"/>
</svg>
```

Here's a paragraph after the figure.
````

Adding a Link Around an Image

You can also use the Markua link syntax around an images itself. This is the standard Markdown link syntax which Markua inherited unchanged.

So, you can do the following:

```
[![A Piece of Cake](cake.jpg)](https://peterarmstrong.com/pictures/1.jpg)
```

Frankly, this syntax has enough line noise to remind me a bit of JWZ's "now you have two problems" joke[9] about regular expressions.

What this is doing is inserting an image as a resource with a caption `![A Piece of Cake](cake.jpg)` and then wrapping the entire thing with a link. If you think of the image resource as `r`, this is just `[r](https://peterarmstrong.com/pictures/1.jpg)`. Simple!

[9] https://blog.codinghorror.com/regular-expressions-now-you-have-two-problems/

Video

The syntax to insert a video is the same compact and consistent syntax that is used for any resource. Local and web resource locations are supported for both video formats; inline resource locations for video are obviously not supported.

Leanpub may choose to not support video at all, or to only support one of local or web video due to bandwidth or copyright concerns.

Unlike images, which are supported in most circumstances, with video files it's currently a bit of a crapshoot. There's currently a dominant proprietary format (H.264, or .mp4) and a new open source challenger (WebM). It's entirely likely that many ebook readers won't support either.

Also, unlike images where many images will accompany the manuscript itself, with video it's expected that authors will be uploading their videos to sites such as YouTube first, and then reusing these videos in the contents of their Markua books. So, while video can be local or web video, web video will be much more prominent than web images in a Markua book.

Furthermore, unlike with web images where the format (and thus the type) are typically discoverable from the file extension in the URL, with web video it's usually not. So, for web videos, there will typically be a `{type: video}` attribute list at a minimum. That said, if the type and format are not specified and the resource is a web resource, Leanpub may use the domain to decide what type of resource to assume. So YouTube videos just work in Leanpub, without specifying either the `type` or `format`.

Per the table of resource types and formats, the following types of videos are supported in Markua: MP4 and WebM.

We will discuss the supported and the default attributes for videos, and then show examples of videos being inserted for both local and web videos.

Supported Attributes for Video

The following are the supported attributes for video resources, in addition to the `type`, `format`, `caption` and `class` attributes which all resources support. Note that the `caption`, `class`, `fullbleed`, `height` and `width` attributes apply to the `poster` image for the video–not to the video itself during playback.

`align`
: The `align` can be `left`, `right` or `middle`. This applies to the `poster` image for the video, and works just like it does with images.

`embed`
: true or false. If true, then you can actually embed the video file or reference it and play it. If false, then it's from a site like YouTube which disallows this. In this case, this should function like a link to external web page, but open an appropriate app (e.g. the YouTube app) instead of a browser. Leanpub can be smart about defaults, and parse the URL to set the value of the `embed` attribute.

`float`
: The `float` can be `left`, `right`, `inside` or `outside`. This applies to the `poster` image for the video, and works just like it does with images. It is an error to specify both `align` and `float`.

`fullbleed`
: `true` or `false`. The default is `false`. This applies to the `poster` image for the video, and works just like it does with images.

`height`
: This applies to the `poster` image for the video, and works just like it does with images.

`poster`
: The `poster` is the URL or path to an image which should be shown instead of the video before the video is played. If Leanpub is outputting some format where it is known that video resources are not supported, it must choose the poster to use as a replacement for the video. Books are not just ebooks–books can also be printed on the fibers of trees that have been chopped down (“paper”), producing something called a “book”. These “books”, whether they are bound in a sturdy fashion (“hardcover books”) or a flimsy fashion (“paperback books”), have one thing in common with respect to embedded video: they do not support it. Finally, if Leanpub is embedding a web video from a web video service (e.g. YouTube) which has a known algorithm for the location of the poster images for its videos, Leanpub may choose to use that poster image if a `poster` attribute is not explicitly specified. However, if a poster attribute is specified, then it must be used by Leanpub.

`poster-format`
: The format of the `poster` image, if it exists. If this attribute is not specified, the format of the `poster` is inferred from the `poster` image file extension. This attribute needs to exist only to support `poster` images which do not have file extensions.

`transcript`
: The URL or path to a transcript of the video, which should be shown or dictated to people who cannot view video. This helps people with visual disabilities view course material, and helps with ADA compliance. For example, what Leanpub does with a `transcript` is as follows: a URL is linked to in the caption (but in a way which does not affect the Table of Contents), and a path is assumed to be a path to a Markua file which is then used to produce a web page at a public URL, which then is also linked to in the caption in a way which does not affect the Table of Contents. With the Markua files for transcripts, Leanpub only supports certain resource types to be included in the transcript files themselves. Specifically, images, video, audio are not supported, but code and math are supported. When Leanpub generates a transcript, the URLs are publicly accessible but obscure. Identical transcripts should not make duplicate transcript files on the web, and using a new UUID every time would violate this. However, not changing the transcript URL when its content changed could lead to bad outcomes. So, the URL of a transcript on the web should be scoped to the book and affected both by its filename and its content. What Leanpub does is: (filename minus extension minus period plus hyphen plus the SHA hash).html. Including the filename eliminates collisions if filenames are unique but the hashes are not.

`width`
: This applies to the `poster` image for the video, and works just like it does with images.

Local Video

```
Here's a paragraph before the figure.

![A Piece of Cake](cake-eating-contest.webm)

Here's a paragraph after the figure.
```

Web Video

```
Here's a paragraph before the figure.

{type: video, poster: "http://img.youtube.com/vi/VOCYL-FNbr0/mqdefault.jpg"}
![Introducing Markua](https://www.youtube.com/watch?t=105&v=VOCYL-FNbr0)

Here's a paragraph after the figure.
```

Ironically, it took me about a year after that talk was recorded to finally figure out what Markua was.

Audio

Just as with video, the audio support in ebooks and on the web is more varied than for images. With audio, there are MP3, AAC, Ogg and WAV formats all in widespread use, and there are a number of other formats with supporters. It's entirely likely that many ebook readers won't support any of them.

Leanpub may choose to not support audio at all, or to only support one of local or web audio due to bandwidth or copyright concerns.

Per the table of resource types and formats, the following types of audio resources are supported in Markua: MP3, AAC, WAV and Ogg Vorbis.

The syntax to insert an audio file is the same compact and consistent syntax that is used for any resource. Local and web resource locations are supported for both audio formats; inline resource locations for audio are obviously not supported.

We will discuss the supported and the default attributes for audio files, and then show examples of audio being inserted for both local and web audio files.

Supported Attributes for Audio

The following are the supported attributes for audio resources, in addition to the `type`, `format`, `caption` and `class` attributes which all resources support.

`transcript`
: The URL or path to a transcript of the audio, which should be shown to people who

cannot hear audio. This helps people with auditory disabilities view course material, and helps with ADA compliance. Transcripts should be produced and handled in an identical way to `video` resources. For example, Leanpub does this. See the Video resource section above for more information.

Local Audio

```
The full version of the talk is here:

![Full Talk](talk.m4a)
```

Web Audio

```
The full version of the talk is here:

![Full Talk](http://markua.com/talk.m4a)
```

Code

Code can be a local, web or inline resource, just like any other resource, and the same resource syntax applies to code as to all other resources.

As discussed, code cannot have alt text. It's just text. If any alt text is provided for a code resource, it is ignored.

Markua specifies only one specific file extension to be associated with a type of `code`: the `.txt` extension, which is for the format of `text`. However, Leanpub interprets **all** unspecified file extensions as specifying a resource of type `code` with a format of `guess`.

Regardless of whether syntax highlighting is supported and the programming language is detected, all code must be formatted as `monospaced text` by Leanpub.

The `text` format means to not do any syntax highlighting as well.

The `guess` format is a request for Leanpub to guess at the programming language based on the file extension and/or the syntax of the code itself. Then, if the detected language corresponds to a particular programming language which Leanpub recognizes, it can format the resource as nicely syntax-highlighted code. Leanpub uses Pygments for syntax highlighting. If Leanpub cannot detect a matching supported programming language, then it formats the code as though the format was `text`–i.e. to format it as unformatted monospaced text.

Besides the `text` and `guess` values of the format attribute, you can also specify the programming language by setting the format attribute to a specific programming language. This is more reliable than `guess`. Unlike other resource types, Markua does not specify the complete set of the values of the `format` attribute–there are so many programming languages in the world, and new ones are added so frequently, that doing so would be impractical.

However, while a complete set of the values of the `format` attribute is not specified, Markua does specify the `console` value of the `format` attribute to indicate console input. Leanpub should format console input as such. (For example, Leanpub uses the open source Pygments library for its code formatting, and Pygments handles `console` format correctly, so Leanpub gets this for free.)

The default value of the format attribute for code is complex:

1. For code which is inserted as a span (which is only supported with inline resources), the default format is `text`.
2. For code which is inserted as a figure which is inserted as an inline resource using three tildes, the default format is `text`.
3. For all other code, the default format is `guess`. This includes local and web resources inserted as figures, and code inserted as an inline figure using three backticks.

Note that the default format can be overridden by specifying it via an attribute list, or after the three backticks in syntactic sugar.

Supported Attributes for Code

The following are the supported attributes for code resources, in addition to the `type`, `format`, `caption` and `class` attributes which all resources support.

`line-numbers`
: This determines whether the code sample shows line numbers. Legal values are `true` or `false`. The default value is `false`. Any value other than `true` is interpreted as `false`.

`number-from`
: If line numbers are shown, this lets you override the starting number of the line numbers. The default value is `1`.

`crop-start`
: Sometimes it's desirable to only show part of a code resource defined in an external file as the code example. The `crop-start` and `crop-end` attributes let you accomplish this. The `crop-start` attribute defines the line which represents the first line included from the resource. For example, `{crop-start: 10, crop-end: 15, line-numbers: true, number-from: 10}` ensures that lines 10-15 are shown and are numbered as lines 10-15. The default value is 1, which is the first line of the file.

`crop-end`
: This attribute ends the range started with crop-start. The default value of crop-end is to be omitted, which is equivalent to specifying the last line of the file.

Default Value of the `format` attribute in Inline Code Samples

The default value of the `format` attribute for a code resource inserted as a figure varies based on context.

If the code resource is a local or web resource, it defaults to `guess`.

If the code resource is an inline resource, the default varies based on the delimiter, and whether the code is inserted as a span or as a block.

With three backticks the default format is `guess`, and with three tildes, the default format is `text`. This way, you can vary the default without having to type an attribute list: if you want the code language guessed at, use backticks; if you don't, use tildes. Of course, you can specify any attributes you wish with either delimiter, and specified attributes override default ones. The only reason there are different defaults are to make things easier to type. Programmers refer to such niceties as "syntactic sugar".

The default value of block code resources inserted with three backticks can be overridden from `guess` to some other value by setting by the `default-code-language` attribute on the entire Markua document. (This attribute has no effect on resources inserted with three tildes.) The default value of code span resources inserted as spans with single backticks can be overridden from the default value of `text` to some other value by setting the `default-code-span-language` on the entire Markua document.

Local Code Resources

Local code resources can be inserted as a figure. In all the following examples of figures, the text in the square brackets is the figure caption, like it is in all figures.

This first figure will be a type of code and a format of `guess`. Leanpub which associates `.rb` file extensions with Ruby code will treat this as Ruby code; Leanpub which has no association for `.rb` files will treat it as plain text:

```
Here's a paragraph before the figure.

![Hello World in Ruby](hello.rb)

Here's a paragraph after the figure.
```

That is equivalent to:

```
Here's a paragraph before the figure.

{format: guess}
![Hello World in Ruby](hello.rb)

Here's a paragraph after the figure.
```

If you don't want to take chances you can do this:

```
Here's a paragraph before the figure.

{format: ruby}
![Hello World in Ruby](hello.rb)

Here's a paragraph after the figure.
```

Note that the caption is optional in all figures:

```
Here's a paragraph before the figure.

![](hello.rb)

Here's a paragraph after the figure.
```

Web Code Resources

Web code resources function identically to how local code resources work, including the significance of file extensions. The only differences is that the files are on the web.

This will be a type of code and a format of `guess` since the file extension is not specified:

```
![Hello World in Ruby](http://markua.com/hello.rb)
```

That is equivalent to:

```
{format: guess}
![Hello World in Ruby](http://markua.com/hello.rb)
```

If you don't want to take chances you can do this:

```
{format: ruby}
![Hello World in Ruby](http://markua.com/hello.rb)
```

Note that in the above examples of figures, the text in the square brackets is the figure caption, like it is in all figures.

Inline Code Resources

Inline code resources are the most flexible way to insert code. They are the only way to insert code as a span resource, and the most straightforward way to add short code examples as figures.

Span

The format of an inline code resource inserted as a span is:

```
Yada yada `some code here` yada yada.
```

The format of code inserted as a span defaults to `text`. However, you can override this by attaching an attribute list to the end of a span.

In the following example, the first statement let the format of the code default to `text`, whereas the second statement specifies the format of the code as `ruby`:

```
Hello World in Ruby is a simple `puts "hello world"` statement.

Hello World in Ruby is a simple `puts "hello world"`{format: ruby} statement.
```

Note that there is no syntactic sugar for this, since I don't like this, and I want it to be as distasteful as possible. So, this is a sort of "syntactic salt". Syntax highlighting does not make much sense for code which is inserted in the middle of a sentence–there's just not enough code there for it to be worthwhile. However, in Markua, any span can have an attribute list attached to it (in order to add index entries, for example), so there is no reason to explicitly forbid specifying a format in the attribute list.

Figure

Inline code resources work great as figures.

This will be a type of code and a format of `guess` since three backticks are used and since the format is not specified:

````
Some paragraph.

```
puts "hello"
```

Some paragraph.
````

That is equivalent to:

````
Some paragraph.

```guess
puts "hello"
```

Some paragraph.
````

If you don't want to take chances you can do this to explicitly specify the format:

```
Some paragraph.

```ruby
puts "hello"
```

Some paragraph.
```

This Ruby code may be formatted as such if Leanpub understands `ruby`. If not, the `ruby` format will be ignored.

If you don't like syntactic sugar you can do:

```
Some paragraph.

{format: ruby}
```
puts "hello"
```

Some paragraph.
```

If you want a figure caption, you can add it to the attribute list with any of the above. For example:

```
Some paragraph.

{caption: "Hello World in Ruby"}
```ruby
puts "hello"
```

Some paragraph.
```

Finally, if you want the code to definitely not get syntax highlighted, you can force format to be `text` in one of two ways.

First, you can set it explicitly:

```
Some paragraph.

```text
puts "hello"
```

Some paragraph.
```

Second, you can use three tildes instead of three backticks, since the default with tildes is `text` not `guess`:

```
Some paragraph.

~~~
puts "hello"
~~~

Some paragraph.
```

Finally, as discussed previously, console input and output should be formatted as such by Leanpub:

```
```console
$ git init
Initialized empty Git repository in /path/to/repo
```
```

Marking Code as Added or Deleted

Markua supports marking code as added or deleted, which can be helpful if you are writing a computer programming book and want to indicate what code should be added or removed to a larger program.

The way to do this is to add special comment lines to your code.

The magic words are `markua-start-insert`, `markua-end-insert`, `markua-start-delete` and `markua-end-delete`. Any line containing one of those words will be removed completely by Leanpub before being inserted into the output.

The Leanpub will then be able to determine which code is being deleted or inserted, and format it accordingly. The recommended way for Leanpub to do this is to make code which is being inserted get `bolded`, and to make code which is getting deleted to be put in ~~`strikethrough`~~.

Finally, while syntax highlighting is optional in Leanpub, if Leanpub does support syntax highlighting it is allowed for Leanpub to not do any syntax highlighting when there is the presence of any of any special `markua-*` comments. Syntax highlighting may make it harder to notice the added and removed code, if they are formatted with bold and strikethrough respectively.

Line Wrapping in Code Resources

Code resources should have newlines added by the author to ensure that automatic line wrapping is not relied upon. Leanpub wraps lines to ensure that all code is visible on a page, and adds backslash `\` continuation characters in the output to indicate that a line has been automatically wrapped.

Math

Math can be a local, web or inline resource, just like any other resource, and the same resource syntax applies to code as to all other resources.

Leanpub currently only supports LaTeX math.

Supported Attributes for Math

The following is the supported attribute for math resources, in addition to the `type`, `format`, `caption` and `class` attributes which all resources support:

`alt` The `alt` is the alt text, to be displayed when the mathematical equations cannot be shown. The default alt text for math is "math". This can be provided in the figure attribute list.

Note that for math, the `format` is the name of the syntax used to write the mathematical equations. There are two special types of `format` for math baked into Markua: `latex` for LaTeX math and `mathml` for MathML math.

Inline Math Resources

Inline math resources are the most flexible way to insert math. They are the only way to insert math as a span resource, and the most straightforward way to add short math examples as figures. LaTeX math, AsciiMath and MathML can be inserted inline as a span or figure.

Span

Being able to insert a math resource as a span is important, as it lets you write things like one of the kinematic equations $d = v_i t + \frac{1}{2} a t^2$ inside sentences. This can be done with LaTeX math.

To insert math as inline math, use a `$` after closing backtick for LaTeX math.

LaTeX math

There is syntactic sugar for LaTeX math which is inserted as a span, using the `$` character after the closing backtick:

```
Here's one of the kinematic equations `d = v_i t + \frac{1}{2} a t^2`$ inside a sentence.
```

The `$` character indicates the inline resource is LaTeX math.

If you don't like syntactic sugar, you can also use after the inline span resource:

```
Here's one of the kinematic equations `d = v_i t + \frac{1}{2} a t^2`{format: latex} inside a sentence.
```

Figure

LaTeX math can be inserted inline as a figure, by specifying either `latex` or `$` after three backticks, or by specifying an attribute list of `{format: latex}`.

All three let you produce mathematical equations like this:

$$\left|\sum_{i=1}^{n} a_i b_i\right| \le \left(\sum_{i=1}^{n} a_i^2\right)^{1/2} \left(\sum_{i=1}^{n} b_i^2\right)^{1/2}$$

LaTeX math

Here’s how you do this using LaTeX math…

Here’s the version with the syntactic sugar for the format after the backticks:

````
Here's a paragraph before the figure.

```$
\left|\sum_{i=1}^n a_ib_i\right|
\le
\left(\sum_{i=1}^n a_i^2\right)^{1/2}
\left(\sum_{i=1}^n b_i^2\right)^{1/2}
```

Here's a paragraph after the figure.
````

Here’s the same thing, with the full format after the backticks:

````
Here's a paragraph before the figure.

```latex
\left|\sum_{i=1}^n a_ib_i\right|
\le
\left(\sum_{i=1}^n a_i^2\right)^{1/2}
\left(\sum_{i=1}^n b_i^2\right)^{1/2}
```

Here's a paragraph after the figure.
````

Here’s the same thing again, with a full attribute list:

````
Here's a paragraph before the figure.

{format: latex}
```
\left|\sum_{i=1}^n a_ib_i\right|
\le
\left(\sum_{i=1}^n a_i^2\right)^{1/2}
\left(\sum_{i=1}^n b_i^2\right)^{1/2}
```

Here's a paragraph after the figure.
````

Tables

Tables are important elements in many Markua documents.

Markua uses the tables that are specified by GitHub Flavored Markdown[10], otherwise known as GFM.

Here's how GFM tables look:

```
| Header 1  | Header 2  |
| --------- | --------- |
| Content 1 | Content 2 |
| Content 3 | Content 4 |
```

GFM tables don't need to line up:

```
| Header 1  | Header 2 |
| --- | --- |
| Content 1 | Content 2 |
| Content 3 | Content 4 Can be Different Length |
```

Finally, to specify column alignment, you add a colons to the header separator row. You still need at least three hyphens per header separator cell:

```
| Left | Middle  | Right |
| :--- |  :---:  |  ---: |
| a    | b       | c     |
| d    | e       | f     |
```

Supported Attributes for Tables

If there are any errors with the supported attributes for tables, such as the `column-widths` not adding up correctly or missing `%` signs, Leanpub must add these errors to a list of errors

[10] https://github.github.com/gfm/#tables-extension-

and warnings that it shows the author, and just output the table with either the erroneous attribute not set or with none of the attributes set.

The following is the supported attribute for table resources, in addition to the `type`, `format`, `caption` and `class` attributes which all resources support:

`column-widths`
: The column widths as a comma-separated list of numbers (integers and/or floats) and/or `*` symbols, from left to right, as a percentage of the total table width. In this attribute value, * means for the column to use the remaining space, equally divided between it and any other column with the * attribute. Examples: `{column-widths: "10%,30%,*,10%"}`, `{column-widths: "10%,*,40%,*"}`, `{column-widths: "10%, 30%, *, 12.5%"}`, `{column-widths: "95%, *, *"}`. The numbers used for the column-widths percentages must sum to exactly 100 (if only numbers are used), or to less than 100 (if there are any `*`s used). Every specified value must be at least 1, and every `*` must compute to at least 1. The number of values (numbers or `*`s) must match the number of columns. Like with the `width` attribute, the percentage sign (%) is required, to make it absolutely clear that these are not measurements in pixels or points. Finally, the list may contain optional whitespace before and/or after each comma.

`width`
: The width of the table, in percentage of page content area width (respecting margins). This is specified as a number (integer or float) between 1 and 100 followed by a percentage sign (%). The quotes are optional. So, you can say `{width: "70%"}`, `{width: 70%}`, `{width: "70.5%"}` or `{width: 70.5%}`.

Leanpub may do whatever it wants when outputting a table. For example, Leanpub may choose to transform the table into an image, for maximum ebook reader compatibility–but at the expense of accessibility support in newer ebook readers.

Whitespace: Spaces, Tabs and Newlines

The goal for the handling of normal whitespace (spaces, tabs and newlines) in Markua is for everything to just work.

There are the four principles of Markua's whitespace handling:

1. You should be able to look at a Markua document and know what is produced. Invisible formatting is frowned upon.
2. Paragraphs and sentences should be handled consistently, regardless of indentation and spaces after periods.
3. Manual whitespace formatting should be discouraged.
4. Newlines are newlines; spaces are spaces. These are different things.

These simple goals have far-reaching consequences:

1. Whitespace at the end of a line or file is ignored.
2. It doesn't matter how many spaces you add after a sentence.
3. All consecutive blank lines after the first blank line are ignored when separating paragraphs, and all consecutive blank lines after the second blank line are ignored when separating lists.
4. You can't manually wrap text with newlines being used as though they are spaces, but you can add forced line breaks without hacks.

Whitespace handling is the largest difference between Markua and Markdown, so it's discussed here, instead of in the appendix.

Newlines

Single Newline = Forced Line Break

In Markdown, you can manually wrap headings, paragraphs, lists and blockquotes with single newlines with no effect on the HTML output. Markdown, like HTML, treats single newlines as equivalent to single spaces.

In Markua, however, a forced line break in the input is a forced line break in all output formats. This is true for paragraphs, lists, blockquotes, asides and blurbs.

In ancient history, some text editors did not automatically wrap lines of text, so manual wrapping of plain text files was a good thing to do. Also, for computer programmers, we

still do not wrap our text when programming. However, for writing, automatic wrapping of paragraphs is essential for staying in the flow while writing, and for being able to edit your text without needing to re-wrap every line in a paragraph. This is one decision that even Microsoft Word gets right.

The decision in Markua to treat single newlines as forced line breaks means that Markua does *not* need to use the **horrible hack** that Markdown uses to output forced line breaks. In Markdown, to output a forced line break (a `<br/>` tag in HTML), you need to add two spaces at the end of the line, followed by a single newline. This means that it is **impossible** to look at a Markdown document with single newlines in it and understand what they mean: you need to find out if there are invisible formatting characters at the end of the line to find out if the newlines mean “newline” or “single space”. Yuck!

Worse, some text editors (like Emacs, the editor I use) can be configured to remove trailing spaces at the ends of lines automatically when a file is saved. So, it’s possible for me to dramatically modify a Markdown document by simply opening it and saving it unedited. Yikes!

The following is an example of Markua’s single newlines:

```
I'm paragraph one. Yay!

This is paragraph two.
This is *still* in paragraph two, preceded by a forced line break.
This is also in paragraph two, also preceded by a forced line break.

This is paragraph three.
```

Three or More Newlines = Two Newlines = One Blank Line

Markua handles two consecutive newlines identically to Markdown: they produce a blank line, which separates block elements like paragraphs from each other.

Similarly, Markua handles three or more consecutive newlines identically to Markdown: they produce one blank line, as though only two newlines had been typed.

If you absolutely must insert a bunch of newlines in a row, you can do this by starting a poetry block and doing so:

```
...the end of a paragraph.

—

—

That empty poetry block is 3 lines long, so it adds three poetic blank lines to the output.
```

One Blank Line Is Added When Concatenating Manuscript Files

A Markua document can be written in one file or multiple manuscript files. If a manuscript is written in multiple files, these files are concatenated together by Leanpub to produce one temporary manuscript file, and that one file is used as the input.

Importantly, in order to avoid a number of bugs, the files are not just concatenated together unchanged–they are concatenated together with **two newlines** (i.e. one blank line) added between the end of each file and the beginning of the next file.

This is needed in order to separate the content of the two files with one blank line between them, in order to prevent a number of surprises for authors. Note that because of this rule, a paragraph (or any other block element) cannot span multiple manuscript files.

All Blank Lines at the Beginning and End of a File are Removed

Since a blank line is added when concatenating multiple manuscript files, there is no good reason to support blank lines, or lines containing only whitespace, at the beginning or end of a file. So, all blank lines and all whitespace-only lines at the beginning or end of a file are removed.

This is especially important with the whitespace at the end of a file: trailing whitespace at the end of a file is invisible to the author, and supporting invisible formatting–whether at the end of a line or the end of a file–is insanity.

Spaces and Tabs

Spaces and Tabs at the Beginning of a Line are Only to Determine List Containment, and Extra Spaces are Removed

Spaces and tabs at the beginning of a line are only used to determine whether the content is contained in a list item–or, in the case of a nested list, which list the list item is contained in.

Besides this, in a paragraph, any manual indentation (using spaces or tabs at the beginning of a line) is just removed. This is even true after a forced line break, using a single newline.

Spaces and Tabs at the End of a Line are Removed

Unlike Markdown, all trailing spaces at the end of a line are ignored by Markua. This way, there is no reliance on invisible formatting to produce newlines, and editors which strip trailing spaces have no effect on a Markua document.

Internal Spaces are Collapsed to One Space, Except At the End of Sentences

Markua handles internal whitespace in a paragraph in a similar way to Markdown:

First, in Markua and Markdown, multiple internal spaces or tabs in the middle of a sentence are all collapsed to one space.

However, Leanpub should be smart about interpreting what is the end of a sentence, and handle that specially.

At the end of sentences that aren't followed by newlines, Leanpub may output one space, one and a half spaces, two spaces or some other amount of space. (Yes, one and a half spaces at the end of a sentence is a real thing, and it is arguably the one true amount of space at the end of a sentence.)

The amount of space chosen to be output at the end of sentences must be output by Leanpub at the end of all sentences which aren't followed by newlines, regardless of whether any given sentence has one or two spaces at its end.

But what's the end of a sentence?

This would be a lot easier to determine if all authors typed two spaces at the end of their sentences! This way, Leanpub could easily determine that something like "Mr. Armstrong" did not, in fact, contain the end of a sentence.

However, many authors type one space at the end of their sentences. So, Leanpub should use heuristics to determine what is the end of a sentence and what is not.

Regardless of how it made the determination about whether a sentence has ended, if Leanpub decides that something is, in fact, the end of a sentence, it must output the same amount of space every time. This can be one space, one and a half spaces, two spaces or some other amount.

Lists

Markua supports two types of lists, bulleted lists and numbered lists, which can be formatted as either simple or complex lists.

The basics of bulleted and numbered lists are discussed first. This is followed by a discussion of simple and complex lists, which will contain examples of both.

Note that Markua distinguishes between “bulleted” and “numbered” lists, not between “unordered” and “ordered” lists (as is done by HTML and Markdown), since all lists have an order–otherwise they wouldn’t be lists!

Bulleted Lists

Markua lets you make a bulleted list by starting each list item with either an asterisk (*) or a hyphen (-), followed by one space, followed by text content. You can’t mix and match asterisks and hyphens in the same list.

You can build a bulleted list out of items starting with an asterisk and one space:

```
* foo
* bar
* baz
```

You can build a bulleted list out of items starting with a hyphen and one space:

```
- one
- two
- three
```

Markua could have supported just one of the asterisk or the hyphen, but this would have been too prescriptive.

To make a bulleted list in Markua:

- Start each list item with either an asterisk (*) or a hyphen (-).
- You can’t mix and match asterisks and hyphens in the same list.
- Only one space is allowed after each bullet. Just as with headings, there is no reason to support any other number of spaces, and the increased consistency is a benefit.
- A single-element bulleted list is a list: although it is a pretty stupid list, treating it as a literal paragraph starting with an asterisk or hyphen would be even stupider.

In terms of style guidance, the preferred bullet type in Markua is the asterisk.

Numbered Lists

In Markua, a numbered list can vary the following:

1. Numbering system
2. Numbering direction (ascending or descending)
3. Initial number (or letter, or Roman numeral)
4. Period or parentheses after the number (or letter, or Roman numeral)

The following choices of numbering system are supported:

1. Decimal numbers
2. Uppercase letters
3. Lowercase letters
4. Uppercase Roman numerals
5. Lowercase Roman numerals

Unlike in Markdown, in Markua the number that begins the list in the manuscript is the *same* number that begins the list in the output.

To make a numbered list in Markua:

- Start two or more consecutive lines with either a consecutive (e.g. `1.`, `2.`, `3.`) or the first (e.g. `1.`, `1.`, `1.`) number, letter or Roman numeral, each followed by either a period or right parenthesis, then exactly one space, then text content.
- Since Markua supports letters and Roman numerals as well as decimal numbers to start lists, the rules about using consecutive numbers or the same number are actually a bit complex. These are discussed later.
- You need to follow the period or parenthesis with exactly one space. Markua very deliberately does not allow more than one space following the period or right-parenthesis: if your list grows to 10 or more items, you should not waste time adding a space to items 1-9 to line their content up with item 10; similarly, if your list grows to 100 or more items you should not waste time adding yet another space to items 1-99 to line their content up with item 100. So, Markua just forbids more than one space after the period or parenthesis. Besides the time saved, the increased consistency is a benefit.

This set of examples shows many of the normal use cases of lists with the various numbering systems. For the edge cases, see the next sections.

You can build a numbered list out of ascending decimal numbers starting from 1:

```
1. foo
2. bar
3. baz
```

You can build a numbered list out of ascending decimal numbers starting from 1, with parentheses used instead of periods:

```
1) foo
2) bar
3) baz
```

You can build a numbered list out of ascending decimal numbers followed by periods starting from a higher number:

```
9. foo
10. bar
11. baz
```

You can build a numbered list out of ascending decimal numbers followed by parentheses starting from a higher number:

```
9) foo
10) bar
11) baz
```

You can build a numbered list out of descending decimal numbers:

```
3. foo
2. bar
1. baz
```

You can build a numbered list out of identical decimal numbers, if you are lazy (producing 1, 2, 3):

```
1. foo
1. bar
1. baz
```

You can build a numbered list out of ascending lowercase letters:

```
a. foo
b. bar
c. baz
```

You can build a numbered list out of ascending lowercase letters, with parentheses used instead of periods:

```
a) foo
b) bar
c) baz
```

You can build a numbered list out of ascending uppercase letters:

```
I. foo
J. bar
K. baz
```

You can build a numbered list out of ascending uppercase Roman numerals:

```
I. foo
II. bar
III. baz
```

You can build a numbered list out of ascending lowercase Roman numerals:

```
i. foo
ii. bar
iii. baz
```

In Markua, It's Hard to Accidentally Make a Numbered List

Markdown has the interesting combination of supporting one element lists and ignoring the number that a list starts with. This means it's possible to inadvertently start a numbered list by beginning any line with a number followed by a period. The example that John Gruber cites[11] is the following:

```
1986. What a great season.
```

In Markdown, this would produce the following single-element numbered list:

```
1. What a great season.
```

In my opinion, is a blatant violation of the Principle of Least Surprise. (By the way, there is a very gross workaround in Markdown: you prefix the period with a backslash. So, you'd write `1986\. What a great season.` to avoid this.)

Now, if Markua supported single-element numbered lists, this would produce a single-element numbered list:

```
1986. What a great season.
```

This wouldn't be as bad as what Markdown does, but it wouldn't be good either! Exactly how stupid it would look would be determined by how numbered lists were formatted by Leanpub, but it certainly would look wrong.

[11] http://daringfireball.net/projects/markdown/syntax

But what to do?

What Markua does is define a number of common-sense use cases which *do* make numbered lists, and then backs out the rule which results. The resulting rule is as follows:

In Markua, a single element numbered list is a numbered list if at least one of the following is true: it uses a parenthesis for its delimiter, it contains a nested list, or it is itself a child of some list item in a list.

So, something like this is not a list in between two paragraphs. Instead, it's three paragraphs, like you would expect:

```
Yada yada yada

1986. What a great season.

Yada yada yada
```

This is true for numbered lists only–single-element bulleted lists are lists. (It's stupid, but the alternative would have been stupider.)

```
Yada yada yada

* This is a list

Yada yada yada
```

So, in Markua, the automatic creation of a numbered list only happens if you have two or more lines starting with consecutive or identical numbers, letters or Roman numerals, followed by a period or right parenthesis, followed by a space, followed by text content.

This still has some possibly incorrect interpretations, but these will be a *lot* more rare. This matters: if you get burned by the automatic list creation, and you feel that you have to think about whether you can start a sentence with a number, then writing in Markua would feel more like programming than writing.

A Single-Element Numbered List With A Parenthesis Is a Numbered List

This **is** a numbered list:

```
Yada yada yada

1) Foo

Yada yada yada
```

This **is** a numbered list:

```
Yada yada yada

1986) Foo

Yada yada yada
```

This is **not** a numbered list:

```
Yada yada yada

1\) Foo

Yada yada yada
```

This is **not** a numbered list:

```
Yada yada yada

1986\) Foo

Yada yada yada
```

This is **not** a numbered list:

```
Yada yada yada

1. Foo

Yada yada yada
```

This is **not** a numbered list:

```
Yada yada yada

1986. Foo

Yada yada yada
```

A Single-Element Numbered List Which Contains A List Is A Numbered List

This is a numbered list:

```
Yada yada yada

1. Foo
   a) foo
   b) bar
   c) baz

Yada yada yada
```

This is a numbered list:

```
Yada yada yada

1986. Foo
      a) foo
      b) bar
      c) baz

Yada yada yada
```

A Single-Element Numbered List Which Is Contained in A List Is A Numbered List

This is a numbered list, including the `a) hello` part at the bottom.

```
Yada yada yada

1. Foo
   a) foo
   b) bar
   c) baz
2. Bar
   a) foo
   b) bar
   c) baz
3. Baz
   a) hello

Yada yada yada
```

This is a numbered list:

```
Yada yada yada

1986. Foo
      a) foo
      b) bar
      c) baz

Yada yada yada
```

Simple Lists

Both bulleted and numbered lists can either be simple lists or complex lists. Whereas the distinction between bulleted and numbered lists was based on the list delimiter, simple and complex lists are distinguished by the indentation and newlines.

Yes, this distinction essentially comes down to the ancient computer science debate of tabs versus spaces.

A simple list uses one tab per nested indentation level.

If you use spaces for indentation, what you are creating is a complex list, which is discussed next.

Besides the formatting rules for bulleted or numbered lists, the rules for a simple list are:

- There are no newlines in list items. Each list item in a simple list is one paragraph with no forced line breaks or blank lines.
- You cannot insert resources with attribute lists in list items. These are inserted after forced line breaks or blank lines, both of which require newlines, and there are no newlines in list items of simple lists.
- There is a maximum of **four** levels of nesting of lists, including the outermost list. (So, you can have three levels of nested lists under each outermost list item.)
- If you try to nest a fifth level of nesting (or more), Leanpub must raise an error.
- The first indentation level is no indentation: it is at the left margin.
- You must use exactly one tab per nested indentation level. (So, the number of tabs you can have is 0, 1, 2 or 3.)
- You cannot use spaces to indent list items. Simple lists only contain tabs.

This is an example of a simple list which has the maximum number (four) of levels of nested list:

```
1. foo
        a) lorem
                i. unus
                        - one
                        - two
                        - three
                ii. duo
                iii. tres
        b) ipsum
        c) dolor
2. bar
3. baz
```

Flat Lists

The examples of bulleted and numbered lists in the Bulleted Lists and Numbered Lists sections above were *flat lists*, which did not contain any indentation.

Every item in a flat list is at the outermost level, and there are no newlines in list items. A flat list is a special case, in that it conforms to all the rules for *both* a simple list and a complex list.

A flat list is so simple that it is not just a simple list, it's also a complex list. *Whoa.*

Complex Lists

Whereas simple lists use tabs for indentation and cannot have newlines within list items, **a complex list uses spaces for indentation levels and can have newlines within list items**.

Using spaces allows for a lot more complex formatting of list items which is still understandable, including lining up nested resources and multiple-paragraph list items. So, these are called complex lists not "spaces lists", since the distinguishing feature is the complexity supported by the spaces, not the spaces themselves.

Besides the formatting rules for bulleted or numbered lists, the rules for a complex list are:

- Complex lists only contain spaces. You cannot use tabs to indent list items or line up content in list items. If you like hitting the tab key, ensure that you set your text editor to convert tabs to spaces.
- Each list item in a complex list can be one or more paragraphs, and each paragraph can contain forced line breaks. Specifically, there can be forced line breaks (formed by single newlines) and blank lines (formed by exactly two consecutive newlines) in list items.
- You can insert resources with attribute lists in list items. These are inserted after forced line breaks or blank lines.

- After any forced line break or blank line, the content (including a nested list) must be lined up with the beginning of the content in the above line using the appropriate number of spaces (not tabs). This rule is discussed further in the Complex List Indentation section below.
- There is a maximum of **four** levels of nesting of lists, including the outermost list. So, you can have three levels of nested lists under each outermost list item.
- If you try to nest a fifth level of nesting (or more), Leanpub must raise an error.
- The first indentation level is no indentation: it is at the left margin.

> Yes, this is one of the reasons why simple lists cannot contain newlines inside list items. If that was allowed, the natural thing to do would be to support using tabs and spaces to align complex content in those list items. However, what this would also result in would be that converting tabs to spaces could break a list: if we allowed tabs to be used to align subsequent paragraphs in a list item, then converting those tabs to spaces could, in certain cases, result in an incorrect number of spaces being used. This would then result in a valid list becoming an invalid list. A common text editor setting which modifies formatting in an invisible way should not be able to break a Markua document, so forcing tab-indented lists to be simple lists ensures that this is the case. This is preferable to either banning tabs (since many people prefer tabs) or to attempting to support complex lists using tabs (which would be both brittle and confusing).

As discussed, if you attempt to nest more than four levels of nesting, Leanpub must raise an error. However, if you violate the indentation rules or newline rules of both simple lists and complex lists, Leanpub must just interpret your list as normal non-list content, such as a paragraph with line breaks. The reason for this is simple: every block element which is not a list violates the rules for lists–otherwise it would be a list! So, the only “this is not a list” property which actually causes an error is an attempt to add more that four levels of nesting to a simple or complex list. Any other mistake just results in the list being not a list, in order to be more permissive about how Markua documents can be formatted.

Complex List Examples

You can do everything with a complex list that you can do with a simple list. This is a complex list which is the equivalent of the nested list from the simple list example above:

```
1. foo
   a) lorem
      i. unus
         - one
         - two
         - three
      ii. duo
      iii. tres
   b) ipsum
   c) dolor
2. bar
3. baz
```

However, you can do a lot more with a complex list as well.

You can add optional blank lines between some or all of the nested lists, just for readability:

```
1. foo

   a) lorem
      i. unus
         - one
         - two
         - three
      ii. duo
      iii. tres

   b) ipsum

   c) dolor

2. bar

3. baz
```

You can make multiple paragraph list items using blank lines, and add forced line breaks using single newlines:

```
1. This is the first paragraph in the first item in the list.

   This is the second paragraph in the first item in the list. It is followed by a nested list.

   a) lorem
      i. unus
         - This is part of the first list item in a nested list.
           This is still part of the first list item in a nested list, with a forced line break.
         - two
         - three
      ii. duo
      iii. tres
```

```
   b) ipsum

   c) dolor

2. bar

3. baz
```

Note that the amount of space to indent is determined by the content, and that is determined by the width of the number:

```
9. This is the first item in the list.
   This is still part of the first item in the list.
   a) lorem
   b) ipsum
   c) dolor
10. This is the second item in the list.
    This is still part of the second item in the list.
    a) lorem
    b) ipsum
    c) dolor
```

This is a numbered list which contains five list items. It includes an inline code resource, a local code resource, and multiple-line list items:

```
1. This is part of the first item in the list.
   ```ruby
   puts "hello"
   ```
   This is still part of the first item in the list.
2. This is the second item in the list.
   {format: ruby}
   ![Hello, World](hello.rb)
   This is still part of the second item in the list.
3. This is the third item in the list.
   This is still part of the third item in the list.
4. This is the fourth item in the list.
   This is still part of the fourth item in the list.
5. This is the fifth item in the list.
```

This is a bulleted list with the same complexity:

```
* This is part of the first item in the list.
  ```ruby
  puts "hello"
  ```
  This is still part of the first item in the list.
* This is the second item in the list.
  {format: ruby}
  ![Hello, World](hello.rb)
  This is still part of the second item in the list.
* This is the third item in the list.
  This is still part of the third item in the list.
* This is the fourth item in the list.
  This is still part of the fourth item in the list.
* This is the fifth item in the list.
```

Neither the numbered or bulleted list above had multiple paragraphs in it, just forced line breaks. This is what those lists look like with multiple paragraphs instead...

This is the numbered list version:

```
1. This is part of the first item in the list.

   ```ruby
   puts "hello"
   ```

   This is still part of the first item in the list.

2. This is the second item in the list.

   {format: ruby}
   ![Hello, World](hello.rb)

   This is still part of the second item in the list.

3. This is the third item in the list.

   This is still part of the third item in the list.

4. This is the fourth item in the list.

   This is still part of the fourth item in the list.

5. This is the fifth item in the list.
```

This is the bulleted list version:

```
* This is part of the first item in the list.

  ```ruby
  puts "hello"
  ```

  This is still part of the first item in the list.

* This is the second item in the list.

  {format: ruby}
  ![Hello, World](hello.rb)

  This is still part of the second item in the list.

* This is the third item in the list.

  This is still part of the third item in the list.

* This is the fourth item in the list.

  This is still part of the fourth item in the list.

* This is the fifth item in the list.
```

To reiterate, single blank lines (two newlines) make paragraphs, and forced line breaks (single newlines) stay in the same paragraph.

```
Here's a paragraph before the list.

1. This is the first paragraph in the first list item. Yay!

   This is a second paragraph in the first list item.

2. The second list item is boring.

3. The third list item has three paragraphs. This is the first paragraph.
   This is still part of the first paragraph.

   Here's the second paragraph in the third list item.

   Here's the third paragraph in the third list item.

Here's a paragraph after the list.
```

Blank Lines in Complex Lists

Single Blank Lines Within Complex List Items

It is legal to add single blank lines in between list items, to separate the list item into multiple paragraphs.

If there is a blank line within a list item, there **must** also be a blank line at the end of the list item.

The reason for this is simple: since paragraphs are separated by blank lines, and since lists support multiple-paragraph list items, you need to add a blank line at the end of a multiple-paragraph list item for symmetry. Otherwise, you can write something ugly like this:

```
1. foo

   bar
2. baz
```

The above example is so ugly it's not a legal Markua complex list, and instead must be interpreted as paragraphs.

The correct way to write it is:

```
1. foo

   bar

2. baz
```

Two Consecutive Blank Lines End A Complex List

Whereas one blank line can be used in between list items for spacing or between other list item content to separate it into paragraphs, two blank lines always end a list and start a new block element, such as a paragraph or a new list.

There is no reason to be able to add two blank lines in between list items. However, there are reasons to wish to have two lists in a row in a Markua document. So, two blank lines in between two list items stops the previous list and starts a new list. There is no need to use some kind of garbage syntax to separate lists–just add an extra blank line.

Note, however, that two blank lines between single-element numbered lists would actually not produce two numbered lists since, again, in Markua single-element numbered lists *are not lists*.

This all sounds a bit complex, but it actually results in behaviour which is as unsurprising as possible to the author, and a lot less surprising than standard Markdown. This is shown by the following examples…

This is one flat list, which is both as simple list and a complex list:

```
* list one item one
* list one item two
* list one item three
```

This is one list. Since there are blank lines, it's a complex list:

```
* list one item one

* list one item two

* list one item three
```

These are two flat lists. Again, flat lists are both simple lists and complex lists:

```
* list one item one
* list one item two
* list one item three


* list two item one
* list two item two
* list two item three
```

These are two complex lists:

```
* list one item one

* list one item two

* list one item three


* list two item one

* list two item two

* list two item three
```

These are two flat lists. The first is a one-element list; the second is a three-element list:

```
* list one item one


* list two item one
* list two item two
* list two item three
```

These are also two flat lists. The first is a one-element list; the second is a three-element list:

```
* list one item one


* list two item one

* list two item two

* list two item three
```

These are two flat lists:

```
1. list one item one
2. list one item two
3. list one item three


1. list two item one
2. list two item two
3. list two item three
```

These are two complex lists:

```
1. list one item one

2. list one item two

3. list one item three


1. list two item one

2. list two item two

3. list two item three
```

This is a normal paragraph followed by a flat list, since single-element numbered lists are not lists. This example shows the correctness of this decision:

```
1986. What a great season.


1. list one item one
2. list one item two
3. list one item three
```

This is also a normal paragraph followed by a flat list, since, again, single-element numbered lists are not lists. This highly-contrived example will confuse anyone reading your manuscript:

```
1. This is actually a paragraph not a list, since single-element numbered lists are not lists.


2. list one item one
3. list one item two
4. list one item three
```

This is also a normal paragraph followed by a complex list, but this more-contrived example would confuse anyone reading your manuscript:

```
1. This is actually a paragraph not a list.


1. list one item one

2. list one item two

3. list one item three
```

Again, this is also a normal paragraph followed by a complex list, but this even-more-contrived example would confuse anyone reading your manuscript:

```
1. This is actually a paragraph not a list.


2. list one item one

3. list one item two

4. list one item three
```

Parsing of Inline Code Resources Inside List Items

When inserting a code resource as an inline resource in a list item, it must be indented to line up with the list item content it is a sibling of.

For example, in the following list, the inline code blocks are indented by three spaces:

````
1. This is part of the first item in the list.

   ```ruby
   puts "hello"
   ```

   This is still part of the first item in the list.

   ```java
   public class Hello {
       public static void main(String[] args) {
           System.out.println("hello");
       }
   }
   ```

2. This is the second item in the list.
````

When the code itself is output, the number of spaces that the list was indented by must be subtracted from the output by Leanpub.

So, when this list is output, in the first example, `puts "hello"` and `public class Hello {` would both start with no indentation, while the ‘ public static void main(String[] args) {‘ line would be output with 4 spaces of indentation, not 7.

Complex List Indentation

The rule for complex list indentation is as follows:

“After any forced line break or blank line, the content (including a nested list) must be lined up with the beginning of the content in the above line using the appropriate number of spaces (not tabs). This rule is discussed further in the Complex List Indentation section below.”

This rule is very strict, and it has a number of consequences.

First, I want to explain why this rule is the way it is.

The goal is to ensure that all complex lists look the same to every author or editor who works on a manuscript, regardless of their tab settings. Complex lists are, well, complex, and being able to reason clearly about them is important. So, the first goal of the formatting rules is to ensure maximum readability. This is why tabs are banned from complex lists.

Next, in terms of the amount of indentation.

The amount of indentation inside a complex list item is actually *not* arbitrary; **it is completely determined by how many spaces you need to line up with the content in the line above the list item**. For a bulleted list this will always be 2 spaces (1 for the `*` or `-`, followed by 1 for the space); for a numbered list this will always be at least 3 spaces (1 or more for the number/letter/Roman numeral, 1 for the `.` or `)`, and 1 for the space).

Now, this has an important and slightly annoying consequence: while nested lists will always line up with the content of the list item, there are certain situations in which nested lists will not line up with each other.

Specifically, there are three of them which are noteworthy:

1. Numbered lists with decimal numbers which go above 9
2. Numbered lists with uppercase or lowercase letters which go above z
3. Numbered lists with Roman numerals

These situations are shown below.

Numbered lists with decimal numbers which go above 9:

```
8. foo
   bar
   a) lorem
   b) ipsum
9. foo
   bar
   a) lorem
   b) ipsum
10. foo
    bar
    a) lorem
    b) ipsum
11. foo
    bar
    a) lorem
    b) ipsum
```

Numbered lists with uppercase or lowercase letters which go above z:

```
y) foo
   bar
   a) lorem
   b) ipsum
z) foo
   bar
   a) lorem
   b) ipsum
aa) foo
    bar
    a) lorem
    b) ipsum
ab) foo
    bar
    a) lorem
    b) ipsum
```

Lists with Roman numerals:

```
i) foo
   bar
   a) lorem
   b) ipsum
ii) foo
    bar
    a) lorem
    b) ipsum
iii) foo
     bar
     a) lorem
     b) ipsum
iv) foo
    bar
    a) lorem
    b) ipsum
v) foo
   bar
   a) lorem
   b) ipsum
```

Now, frankly, the Roman numeral example looks pretty bad. However, nesting anything inside Roman numerals looks bad. For example, if complex lists in Markua worked with a constant amount of spaces, then they would not line up with the content of the list items.

```
This is not a Markua list, and will be output a really ugly paragraph with a bunch of forced line breaks:

i) foo
   bar
   a) lorem
   b) ipsum
ii) foo
    bar
   a) lorem
   b) ipsum
iii) foo
     bar
   a) lorem
   b) ipsum

Again, the above is not a list.
```

Definition Lists

Definition lists are also supported in Markua. Definition lists are related to lists, but they are neither flat, simple nor complex. Instead, they are what they are: definition lists.

To define a definition list in Markua, use the following syntax:

```
term 1
: definition 1

term 2
: definition 2
```

There can be one to three spaces after the colon, or one tab.

A definition list can provide multiple definitions for a term:

```
term 1
: definition 1a
: definition 1b

term 2
: definition 2
```

A single term definition list is a definition list, regardless of how many definitions for the term exist:

```
term
: definition
```

Finally, like list items in complex lists, each definition list item can contain newlines and multiple paragraphs. What you do here is indent the subsequent lines by the same amount of space as the initial line. (If you do not indent the subsequent lines, then you're ending the definition list and just starting a new paragraph.) As with list items, one newline is a forced line break; two newlines is a new paragraph:

You can nest resources inside a definition list. You cannot nest definition lists inside definition list items, however–that would be highly confusing.

```
Here's a paragraph before the definition list.

one
: This is the first definition of one. It's one paragraph.
: This is the second definition of one. It's two paragraphs.

  This is a second paragraph in the second definition of one.

two
: The second definition list item is simple.

three
: The third definition list item has three paragraphs.
  This is definition still part of the first paragraph.

  Here's the second paragraph in the third definition list item.

  Here's the third paragraph in the third definition list item.

ruby
: Here is some Ruby code.

  ```ruby
  puts "hello"
  ```

  Here is some more Ruby code.

  {format: ruby}
  ![Hello, World](hello.rb)

  That's as simple as it gets.

Here's a paragraph after the list.
```

Just as with list items, any leading whitespace after the line break is used to continue the definition list item, and is thus ignored. Like list items, definition list items are not poetry.

With the rise of mobile and the narrower screen reading experience becoming the new default, definition lists may end up being more useful than tables in many Markua documents.

Finally, with definition lists, one thing you often want to do is link to a specific definition, not just to the list itself. This is useful to do, since in a document with many definitions, it's helpful if the reader scrolls to the right spot or opens to the right page.

To do this, just define a span id on the element itself, and then link to it.

```
foo{#foo}
: This is foo.

bar{#bar}
: This is bar.
```

Note that if you define an id attribute above the first definition list item, what you are doing is defining an id on the entire definition list. As such, this does not work on any subsequent list item: in the following definition list, the `{#definitions}` is the id of the definition list. It is not the id of foo.

```
{#definitions}
foo{#foo}
: This is foo.

bar{#bar}
: This is bar.
```

To be clear, the following is not legal Markua:

```
This is not legal.

foo
: This is foo.

{#bar}
bar
: This is bar.

This is not legal.
```

If there was an id above foo, it would be legal, since it would be the id of the entire definition list. However, the id attribute above bar is not legal. Leanpub should ignore this id, and add an error to the list of errors.

Block Elements

Broadly speaking, Markua documents consist of three things: block elements, span elements and metadata. Paragraphs, headings, figures and lists, all discussed earlier, are examples of block elements. Block elements are separated from each other by at least one blank line.

These are the remaining block elements defined by Markua.

Scene Breaks (`* * *`)

In fiction, scene breaks are sometimes added between paragraphs in a chapter to denote a break in context. To add a scene break, add three or more asterisks on a line by themselves, with or without spaces between them. For example, `***`, `* * *` and `**********` on a line by themselves all produce a scene break.

Example:

```
This is before the scene break.

* * *

This is after the scene break.
```

Blockquotes (`>`)

Blockquotes in Markdown are created by prefacing lines with `>` , i.e. a greater than character followed by a space:

```
This is the first paragraph.

> This is a blockquote.
>
> It is outside the paragraphs.

This is the second paragraph.
```

Blockquotes in Markua are created in one of two ways:

1. By prefacing lines with `>` , i.e. a greater than character followed by a space.

2. By wrapping the blockquote in `{blockquote}` ... `{/blockquote}`

Option #1 is preferable for short quotes; option #2 is easier on authors for really long quotes.

Like figures and tables, blockquotes can be inserted in the middle of a paragraph or as a sibling of it.

> For non-programmers: I'm calling these things "blockquotes", not "block quotes", since their origin is in Markdown blockquotes[a], and since they can be inserted by typing `{blockquote}`. If I called them "block quotes", that might encourage someone to incorrectly try to insert them as `{block quote}`, which would not work.
>
> [a] http://daringfireball.net/projects/markdown/syntax#blockquote

These Markua blockquotes are siblings of the paragraphs:

```
This is the first paragraph.

> This is a blockquote.
>
> It is outside the paragraphs.

This is the second paragraph.

{blockquote}
This is a blockquote.

It is outside the paragraphs.
{/blockquote}

This is the third paragraph.
```

These Markua blockquotes are nested in the paragraph:

```
This is the first paragraph.

This is the second paragraph.
> This is a blockquote
>
> It is inside the second paragraph.
This is part of the second paragraph.
{blockquote}
This is a blockquote.

It is inside the second paragraph.
{/blockquote}
This is part of the second paragraph.
```

```
This is the third paragraph.
```

A blockquote can contain other block-level elements, most commonly paragraphs.

If you are using the `{blockquote}` ... `{/blockquote}` approach, this is trivial: just pretend you're in a normal paragraph, and the syntax is the same.

If you are using the Markdown approach of `>`, then to start a new block level element within a blockquote, just put a line starting with a `>` followed by a space, followed by the block level element. It is equivalent to placing a `>` and a space in front of every line of the paragraphs.

In Markdown, a single newline inside a blockquote (where both lines are preceded by a `>` and a space) adds a single space. In Markua, however, a single newline inside a blockquote adds a forced line break. This is identical to how single newlines inside a normal Markua paragraph function. This is discussed at length in the Single Newlines section earlier. Note that it means you **cannot** manually wrap blockquotes to make them look nicer. Manually wrapping blockquotes is tedious and discourages editing of your own work. If you have really long blockquotes which span multiple paragraphs, the `{blockquote}` syntax is more pleasant to write in.

Blockquotes can be multi-paragraph. To create a multi-paragraph blockquote, you need to separate each paragraph with a line containing a `>` and (optionally) whitespace.

If a blockquote contains headings, these headings may be formatted by Leanpub differently than normal headings. Finally, if Leanpub is automatically generating a Table of Contents from chapter and section headings, any headings inside blockquotes should be ignored.

A blockquote can have a citation. This is done via the attribute list, which can include a `cite` attribute with the text of the citation and a `url` attribute with the URL of the citation. If both are specified, the text of the citation is linked to the URL. If only the `cite` attribute is specified it is shown as text; if only the `url` is specified it is inserted as text with a link to itself.

The attribute list can be used regardless of which syntax is used to insert the blockquote:

```
Lots of people have opinions about software.

Here's the most famous recent one:

{cite: "Marc Andreessen", url: "http://www.wsj.com/articles/SB10001424053111903480904576512250915629460"}
> Software is eating the world.

It's quoted a lot, so let's quote it again:

{blockquote, cite: "Marc Andreessen", url: "http://www.wsj.com/articles/SB10001424053111903480904576512250\
915629460"}
Software is eating the world.
{/blockquote}

That's it!
```

Asides (`A>` or `{aside}`)

Since Markua is for writing books, including technical books, it needs not just a syntax for blockquotes–it also needs a syntax for asides and for blurbs. These syntaxes are very similar to the Markua syntax for blockquotes. Like blockquotes, any headings inside asides or blurbs do not show up in a Table of Contents (if one is present).

We will consider asides first.

Asides are typically short or long notes in books which are tangential to the main idea–sort of like footnotes, but in the body text itself. In technical books, quite often they are formatted in a box. Asides can span multiple pages.

The syntaxes for asides are very similar to blockquotes:

1. By prefacing lines with `A>` , i.e. an `A`, then a greater than character (`>`), then a space.
2. By wrapping the aside in `{aside}` ... `{/aside}`

Option #1 is preferable for short asides; option #2 is easier on authors for really long asides.

For consistency with blockquotes, asides can be siblings of paragraphs or nested in them.

Here's a short aside:

```
A> This is a short aside.
```

Here's a longer aside, which also contains a heading:

```
A> # A Longer Aside
A>
A> This is a longer aside.
A>
A> It can have multiple paragraphs.
A>
A> The `A> ` stuff can get tedious after a while.
A>
A> This is why the `{aside}` syntax exists.
```

Here's a longer aside using the `{aside}` syntax, which also contains a heading:

```
{aside}
# A Note About Asides

This is a longer aside.

It can have multiple paragraphs.

Asides can also have headings, like this one does.

Multi-paragraph asides are more pleasant using this syntax.
{/aside}
```

Blurbs (`B>` or `{blurb}`)

Blurbs are like asides, but shorter. A blurb is not intended to span multiple pages of output.

The syntaxes for blurbs are very similar to asides:

1. By prefacing lines with `B>` , i.e. a `B`, then a greater than character (`>`), then a space.
2. By wrapping the blurb in `{blurb}` ... `{/blurb}`

Examples:

```
B> This is a short blurb.
```

```
B> # A Longer Blurb
B>
B> This is a longer blurb.
B>
B> It can have multiple paragraphs.
```

```
{blurb}
#A Longer Blurb

This is a longer blurb.

It can have multiple paragraphs.
{/blurb}
```

Supported Attributes for Blurbs

Blurbs also have support for an attribute list, which can contain a `class` attribute as well as other implementation-specific “extension attributes”.

Blurb `class` Types

Markua has its origin in authoring computer programming books. In computer programming books, there are a number of blurb types which are a de facto standard:

- `center`
- `discussion`
- `error`
- `information`
- `tip`
- `warning`

These blurb types can be set on a blurb as its `class` attribute. Leanpub can optionally style these blurbs appropriately based on the class, for example by adding custom icons for each class of blurb.

Here's how this looks with the `B>` syntax:

```
{class: warning}
B> This is a warning!
```

Here's how this looks with the `{blurb}` syntax:

```
{blurb, class: warning}
This is a warning!
{/blurb}
```

> The attribute list must either directly precede the `B>` with no blank line between it and the `B>`, or it must be combined with the `{blurb}` block opening. It is not legal Markua syntax to have an attribute list preceding a `{blurb}` block opening like this:
> `{class: warning}`
> `{blurb}`

Syntactic Sugar for Specific Blurb Classes: `D>`, `E>`, `I>`, `Q`, `T`, `W>`, `X>`

Having to constantly type `{class: warning}` in a computer programming book with a number of warnings would get tedious, as would any of the other blurb classes listed above.

So, Markua defines a standard shorthand syntax for these classes of blurbs. With this syntax, you use a different letter than `B` in the `B>`, to create a blurb with the appropriate `class`.

These are the syntactic sugar values you can use which have a heritage in computer programming books:

Sugar	Equivalent To a B> With
D>	{class: discussion}
E>	{class: error}
I>	{class: information}
Q>	{class: question}
T>	{class: tip}
W>	{class: warning}
X>	{class: exercise}

Examples:

```
D> This is a discussion blurb.

E> This is an error blurb.

I> This is an information blurb.

Q> This is a question blurb.

T> This is a tip blurb.

W> This is a warning blurb.

X> This is an exercise blurb.
```

These are equivalent to:

```
{class: discussion}
B> This is a discussion blurb.

{class: error}
B> This is an error blurb.

{class: information}
B> This is an information blurb.

{class: question}
B> This is a question blurb.

{class: tip}
B> This is a tip blurb.

{class: warning}
B> This is a warning blurb.

{class: exercise}
B> This is an exercise blurb.
```

These are *also* equivalent to:

```
{blurb, class: discussion}
This is a discussion blurb.
{/blurb}

{blurb, class: error}
This is an error blurb.
{/blurb}

{blurb, class: information}
This is an information blurb.
{/blurb}

{blurb, class: question}
This is a question blurb.
{/blurb}

{blurb, class: tip}
This is a tip blurb.
{/blurb}

{blurb, class: warning}
This is a warning blurb.
{/blurb}

{blurb, class: exercise}
This is an exercise blurb.
{/blurb}
```

Note that `Q>` and `X>` are a bit controversial:

- `Q>` defines a blurb which is formatted like a question, but `{quiz}` (discussed later) defines a quiz, and quizzes have actual numbered questions in them. It is unfortunate that the words `quiz` and `question` both start with the letter `Q`, and that the `question` blurb is not the same thing as a question in a quiz.
- `X>` defines a blurb which is formatted like an exercise, but `{exercise}` (discussed later) defines a structured exercise similar to a quiz. It is unfortunate that the term "exercise" is used for both.

There are issues, in both cases. However, the alternative is worse: removing the `Q>` or `X>` syntax would cause issues for every author who is familiar with, or has a manuscript formatted in, Leanpub Flavoured Markdown. This is not worth the reduced functionality, just to avoid one possible bit of confusion and one naming collision. So, the `Q>` and `X>` blurb syntactic sugar do exist, as do the `{class: question}` and `{class: exercise}` full blurb syntaxes.

Also note that nothing in this section defines what Leanpub must *do* with the given class of blurb. Leanpub, for example, uses it to add an appropriate icon from Font Awesome[12] at the left of the blurb.

[12] https://fortawesome.github.io/Font-Awesome/

Finally, note that specifying a class in metadata overrides what the syntactic sugar does:

```
{class: tip}
W> This is a tip blurb, not a warning blurb.
```

Leanpub is free to either override this silently, or to raise an error if this happens.

Using Blurbs to Center Text with `C>`

You can also use a blurb to center text.

The following two ways to do this are equivalent:

```
C> This is a centered blurb.

{class: center}
B> This is a centered blurb.
```

This is the only way to center text in Markua.

Unlike other blurb types which have their origin in technical books, centering text has a wide range of uses. So, it could have been thought of as something different than a blurb. However, in terms of its behaviour and the way it's inserted, centered text is a blurb– whether it's inserted via syntactic sugar or via a class attribute on a normal blurb element. So, it's discussed here.

Using Extension Attributes on Blurbs to add `icon` Support

Leanpub adds an `icon` attribute to blurbs. Markua does not specify that a blurb must support an `icon` attribute, or what it would mean if it did. However, Leanpub understands an `icon` attribute to reference an icon from Font Awesome. The value of this attribute is assumed to be the name of an icon in Font Awesome, without the `fa-` prefix. So, in Leanpub, you can do this:

```
{icon: car}
B> You can't spell carbon without it!

{icon: leanpub}
B> Yes, we're in Font Awesome!

{icon: github}
B> So is GitHub, of course. Unicorns.
```

In Leanpub, this will produce a nice icon of a car, using the Font Awesome icon.

Quizzes and Exercises

The final two block elements that Markua provides are quizzes and exercises. These two block elements are very special, however, in that they enable a single Markua document to construct everything from traditional textbooks and paper-based quizzes to entire online courses (or MOOCs). So, they're discussed in their own chapter.

Quizzes and exercises are essentially the same. The only difference is that quizzes are intended to be marked, and exercises are not. Because of their similarities, they are discussed here together.

Quizzes or exercises in a textbook consist of two things:

1. Questions, typically in the chapter itself.
2. Answers, typically at the back of the book.

The questions in the chapter essentially are placed there like any other block element, such as an aside or blurb. The answers are positioned at the back of the book, along with other elements like the index and appendices. The specific location that they are positioned can be controlled by the author using book section directives, discussed later.

There is only one syntax to create a quiz or exercise. For a quiz, it's by wrapping the quiz in `{quiz}` ... `{/quiz}`; for an exercise, it's by wrapping the exercise in `{exercise}` ... `{/exercise}`.

Here is a brief example of a quiz:

```
{quiz, id: quiz1}
? How many letters are in the word Markua?

a) 5
B) 6
c) 7

? How many unique letters are in the word Markua?

! 5
{/quiz}
```

This quiz contains two questions: a multiple-choice question where the correct answer is b, and a fill-in-the-blank question where the correct answer is 5. Quizzes and exercises have the same question types, discussed later.

With a quiz, the `id` attribute is **required**. This is so the identity of a quiz can be preserved across generations of a course.

Here is the same example, but as an exercise:

```
{exercise, id: exercise1}
? How many letters are in the word Markua?

a) 5
B) 6
c) 7

? How many unique letters are in the word Markua?

! 5
{/exercise}
```

Just like with quiz, with an exercise the `id` attribute is **required**. This is so the identity of an exercise can be preserved across generations of a course.

Quiz and Exercise Headings and Other Content

A quiz or exercise can contain any Markua content, not just questions and answers. This is true regardless of whether the quiz or exercise is in a MOOC, an ebook or on paper. Note that video and audio resources don't work so well on paper, however.

If a quiz or exercise starts with any type of heading immediately after the `{quiz}` or `{exercise}` line, this heading's content should be considered the name of the quiz or exercise. This can be used in a list of quizzes or exercises produced by Leanpub. Typically the heading will be a chapter heading (`# `), but section headings (`## `) and lower headings also are supported. (The reason for this is that quizzes are sometimes top-level things, and other times are nested inside chapters, sections or sub-sections. Some course authors would correctly feel that the quiz should have the appropriate level of heading given their position in the document.

Example:

```
{quiz, id: quiz2}
# Markua Quiz

Watch this [video](https://www.youtube.com/watch?time_continue=1&v=VOCYL-FNbr0) of Peter explaining Markua.

? What year was that video from?

What year? Really? Did it really take that long? What was going on???

a) 2012
b) 2013
C) 2014
d) 2015

{words: 500}
? Why do you think the first version of the Markua spec took so long?
```

```
Look at the Leanpub website and read the [pricing essay](https://leanpub.com/pricing).

! Answers could include "bootstrapped startup", the spec evolving, removing HTML mapping, etc.

That's it for this quiz, and this MOOC!

**Thanks for taking my course!**
{/quiz}
```

An Empty Quiz or Exercise is Not an Error

A quiz or exercise which contains no questions is not an error. Instead, a if Leanpub encounters a quiz or exercise with no questions it must filter the quiz or exercise from the output, optionally providing a warning to the author.

This lets authors create placeholders for quizzes or exercises in their courses before the quizzes or exercises are ready, which is potentially very useful in an in-progress course.

A Malformed Quiz or Exercise is an Error

If Leanpub encounters a malformed quiz or exercise it must treat this as an error and not generate the output from the Markua document. Quizzes and exercises are not something that should ever be produced in a broken state.

However, it is also an error to parse quiz syntax outside a quiz or exercise block. Leanpub must not parse lines starting with ? or ! as representing questions or answers unless those are contained in a quiz or exercise block.

Supported Attributes on Quizzes and/or Exercises

`attempts`
: The number of allowed attempts on a quiz. The default is defined by the value of `default-quiz-attempts` on the containing course, or `1` if this is not present. A value of `0` means the quiz cannot be taken (yet). A value of `-1` means the quiz has an unlimited number of attempts. Since an exercise does not count toward the mark on a course, an exercise always has an unlimited number of attempts.

`case-sensitive`
: `true` or `false`. The default is `true`. This sets the default behaviour of fill in the blank questions. If `true`, the fill in the blank question answers are case-sensitive. If false, they are not.

`id`
: All Markua elements support an `id` attribute. The reason the `id` attribute is explicitly listed here is to emphasize that Leanpub may require an `id` attribute on a quiz or

exercise. For example, Leanpub requires the `id` attribute on all quizzes, in order to determine the identity of quizzes when a course is being published in-progress. (As a student, you'd be pretty unhappy if you had to re-take an unchanged quiz simply because a professor published a new course version.)

`points`
: If present, this is the total number of points the quiz or exercise is worth. (This really only matters for quizzes, but is supported for exercises as well, in case Leanpub wishes to display the points on exercises to make them feel more real.) If `points` is not present, the worth of the quiz is determined by summing the points of the questions. (Questions are worth 1 point each if they have no `points` attribute.) If the quiz has a `points` attribute and its questions also have `points` attributes, the worth of each question in a larger course context is determined as follows: its `points` are the numerator, and the total `points` in the quiz or exercise is the denominator.

`random-choice-order`
: `true` or `false`. The default is `false`. This sets the default behaviour of multiple choice questions. If `true`, the choices in the multiple choice question are randomly arranged; if `false`, they are presented in the order written.

`random-question-order`
: `true` or `false`. The default is `false`. This sets the default behaviour of the quiz or exercise. If `true`, the questions are randomly arranged; if `false`, they are presented in the order written.

`start-at`
: The `start-at` is the number of the first question. The default is 1. Any integer is permitted. Subsequent questions will have a number which is 1 higher than the previous question.

`version`
: The version of the quiz. This does not replace the function of the `id`; it's more for use in analytics by the instructor. The default is `1`.

As discussed above, there is no `title` or `caption` attribute for a quiz–you can just add a heading inside the quiz or exercise itself, using the normal Markua formatting for a chapter heading.

Question Types: Multiple Choice, Fill In The Blank

There are two types of questions supported by Leanpub:

1. Multiple Choice
2. Fill In The Blank

These types are not specified by a `{type}` attribute. Instead, they are inferred from properties of the answers or from other attributes of the question.

Multiple Choice Questions

A multiple choice question has 2 or more answer choices, and 1 correct answer choice.

The correct answer choice is in capital letters before the parentheses; incorrect answer choices have lowercase letters before the parentheses.

Example:

```
? How many letters are in the word Markua?

a) 5
B) 6
c) 7
```

Obviously, when generating the question in the actual quiz or exercise, Leanpub must make all answer choices have the same type of letter. This is usually a lowercase letter, although either all lowercase or all uppercase letters would be fine.

Unless a `choose-answers` attribute is used, the multiple choice answers all must start from a or A, and must use a right-parenthesis after the a or A. Any line starting with a) ar A) in a quiz is considered a set of multiple choice quiz answers, not a numbered list using a) or A) as a delimiter. If you want to put a numbered list in a quiz body, use periods for the delimiter.

A multiple choice question may also have a dynamic number of answer choices, including for the correct answer. This done with the special `choose-answers` attribute, shown and explained below.

```
{choose-answers: 4}
? How many grams are in a pound?

C) 454
C) 453
m) 451
m) 1000
o) 100
o) 150
o) 200
o) 250
o) 300
o) 500
```

The `choose-answers` attribute specifies how many answer choices should be shown. This includes exactly one of the correct answers (indicated with `C`), all of the mandatory incorrect answers (indicated with `m`) and as many of the optional incorrect answers (indicated with `o`) as are needed for the question to have the total number of answers as indicated by the `choose-answers` attribute.

So, in the above example, either `453` or `454` will be shown, along with the mandatory incorrect answer choices `451` (a literary joke) and `1000` (a kilogram, not a pound) and one of the optional incorrect answers (`100`, `150`, `200`, `250`, `300` or `500`).

When a `choose-answers` attribute is used, the question will always have `random-choice-order`.

The following are errors in a question where a `choose-answers` attribute is used:

- 0 correct (`C`) answers
- not enough mandatory (`m`) incorrect or optional (`o`) incorrect answers for the question to have the `choose-answers` number of answers
- if `choose-answers` is n, a number of mandatory (`m`) incorrect answers >= n (since there needs to be one correct answer shown)
- if `choose-answers` is n and the number of mandatory (`m`) answers is n - 1, then any optional (`o`) incorrect answers existing
- answers starting with something other than `C`, `m` or `o`

Supported Attributes on Multiple Choice Questions

`choose-answers`
: This is described above. If `choose-answers` is used, `random-choice-order` is forced to true.

`points`
: The number of points the question is worth. This number can be 0 or higher. The default is 1.

`random-choice-order`
: `true` or `false`. The default is `false`, unless `choose-answers` is used. This sets the behaviour of the specific multiple choice question. If `true`, the choices in the multiple choice question are randomly arranged; if `false`, they are presented in the order written. If this attribute is omitted, its value is determined by the `random-choice-order` attribute on the quiz itself, which defaults to `false` if absent.

Fill In The Blank Questions

A fill in the blank question consists of a question and a set of answers. Each answer is specified by `!`, an optional points value, a space, and then a semicolon-separated list of the acceptable values of that answer. Each answer value can be a text string (quoted or not) or a regular expression (regex). If a points value is not specified for an answer, the answer is worth full points.

Support for regular expression answer values is optional. However, Leanpub which supports regular expression marking must interpret any answer which starts with a forward slash (`/`) and ends with a forward slash followed by some word characters (e.g. `i`) as being a regular expression. Note that Leanpub uses Ruby regular expressions.

Finally, note that you can separate regular expressions with semicolons, just like any other answer value. There's no reason not to support this, and it may lead to simpler regular

expressions. However, if you're good at regular expressions, you can also combine them into one regular expression, of course.

Note that since a semicolon is used to separate answer values, to provide an actual semicolon as part of an answer value you must either put the answer value in quotes, use a backslash-escape `\;` or make the semicolon part of a regular expression.

Examples:

```
? How many unique letters are in the word Markua?

! 5

? What's the global capital of investment banking?

! New York ; London

? What's the global capital of investment banking?

! "New York" ; "London"

? What's the global capital of investment banking?

! New York
! London

? What's the global capital of investment banking?

! "New York"
! "London"

{case-sensitive: false}
? What's pi?

! "The ratio of a circle's circumference to its diameter" ; 3.14 ... 3.1416 ; an irrational number

{case-sensitive: false}
? What's pi?

! "The ratio of a circle's circumference to its diameter"
! 3.14 ... 3.1416
! an irrational number

? Where's the Eiffel Tower?

! /(Paris|France)/i

? Where's the Eiffel Tower?

! /Paris/i ; /France/i

{points: 2}
```

```
? Where's the Eiffel Tower?

! /Paris/i
! /France/i

{points: 2}
? Where's the Eiffel Tower?

!2 /Paris/
!1 /paris/i
!.5 /France/i

{points: 2}
? Where's the Eiffel Tower?

! /Paris/
!1 /paris/i
!.5 /France/i

{points: 2}
? Where's the global capital of investment banking?

!2 New York ; London
!1 USA ; UK
```

As shown by the answer (`"The ratio of a circle's circumference to its diameter" ; 3.14 ... 3.1416; an irrational number`), acceptable answer values in a fill in the blank question can be of completely different types, and numeric answer values can be expressed as ranges (min <= x <= max), expressed as `min ... max`. Also, this answer shows that quotes are optional around text strings. The reason to use quotes is for clarity, or to ensure that any semicolons used are treated as semicolons instead of as answer choice delimiters. Semicolons inside quotes are just semicolons and do not need to be backslash-escaped. You do, however, need to backslash-escape a quote if you want it to be treated as a literal quote, instead of the start or end of a string.

Supported Attributes on Fill In The Blank Questions

`points`
: The number of points the question is worth. This number can be 0 or higher. The default is the 1. The answers must either not specify points (in which case they are worth the full value of points that the question is worth), or they must specify points between 0 and the `points` value.

`case-sensitive`
: `true` or `false`. The default is `true`. This sets the behaviour of the specific fill in the blank question. If `true`, the fill in the blank question answer is case-sensitive. If false, it is not. In the case of multiple acceptable answer values, this attribute applies to all of them. Note that this only applies to text string answers, not to regular expressions. For a regular expression to be case-insensitive, you must end it with an `i` after the closing backtick.

Creating a Course or MOOC from a Markua Document

Over the past decade, there has been a steady growth of interest in courses delivered over the internet at massive scale. These Massive Open Online Courses, or MOOCs, consist of essentially four things:

1. Reading material
2. Video or audio lectures
3. Exercises, with answers provided to the student
4. Quizzes, with answers used to automatically mark the quiz

It turns out the four things in this list all work perfectly in a Markua document. So, not only can Markua be used to easily create a textbook which includes video, audio, images and quizzes, it is also an amazingly simple and flexible way of creating a MOOC. A MOOC is essentially just a textbook which is executable, plus discussion forums and credentials. For example, Leanpub[13] authors can click one button to create a massive open online course (MOOC), complete with automated marking for all the quizzes in the course, entirely from one Markua source document.

The fact that a Markua document can be used to create an online course or MOOC means that certain aspects of the syntax for quizzes and exercises are more robust than they would otherwise. One example of this is question alternates.

Question Alternates

In an online course or MOOC, some professors might not want every question the same, despite the fact that question order and answer order can be randomized. So, Markua supports question alternates, using a simple (if slightly ugly) syntax. Question alternates are only supported in quizzes, since they make no sense to include in exercises.

To create question alternates, every question in the quiz (not just those with alternates) must be numbered sequentially, starting from 1, using a `?#` syntax. This is a question mark followed by the number of the question, e.g. `?1, ?2, ?3`. The questions in a quiz are numbered using sequential positive integers starting from 1: 1, 2, 3, etc.

The alternates are specified by providing the same number for multiple questions, e.g. `?1, ?1, ?1, ?2, ?3, ?4, ?4, ?5`. When the actual quiz is given, only one of the questions for the given question number is used.

Note that only the first question with a given number may have a `points` attribute–since all other alternates must use the same points value, specifying it would be pointless.

[13] https://leanpub.com

The following is an example of a quiz which uses question alternates. This ensures that to ensure that students get randomly selected versions of questions 1 and 4. Also, since `random-question-order: true` is used, the actual position of the questions is randomized after the specific questions are selected from the alternates.

```
{quiz, id: "midterm", random-question-order: true}
?1 What's 2 + 2?

! 4

?1 What's 2.2 + 2.2?

! 4.4

?2 what's 3 + 3?

! 6

?3 What's 4 + 4?

! 8

{points: 2}
?4 What's 5 + 5?

! 10

?4 What's 6 + 6?

! 12

?5 What's 7 + 7?

! 14
{/quiz}
```

Note that the syntax for question alternates is very strict. Every question must have a number, and these numbers must be in ascending order (except for the alternates, which have the same number as each other).

Question alternates can also be grouped by a `choose-questions` attribute attached to the first question alternate. In this case, Leanpub must choose the number of questions `m` specified from the given alternates with that number `n`, or `n choose m`. Note that in this scenario, the numbering after the alternates increases by `m`: for example, if a quiz starts with a `choose: 3`, the next question is numbered 4, not 2. This ensures that the person constructing the quiz knows what they are doing, and saves them from having to keep track in a scenario where there are multiple questions with a `choose-questions` attribute.

```
{quiz, id: "midterm", random-question-order: true}

{choose-questions: 3}
?1 What's 2 + 2?

! 4

?1 What's 2.2 + 2.2?

! 4.4

?1 what's 3 + 3?

! 6

?1 What's 4 + 4?

! 8

?1 What's 5 + 5?

! 10

{points: 2}
?4 What's 6 + 6?

! 12

?5 What's 7 + 7?

! 14
{/quiz}
```

Leanpub must treat any error in the numbering of question alternates (and the questions which follow) as an error, and not generate the quiz if there is any error. This is preferable to Leanpub of trying to guess at what the author meant, and trying to do the right thing. Fixing a syntax error takes a couple minutes of editing and a few minutes to publish the book or course again. However, fixing the consequences of a quiz being administered to hundreds–or thousands, or tens of thousands–of people with an incorrect number of questions, or with questions incorrectly used as alternates for each other, would be much more difficult.

Span Elements

We've already seen many examples of span elements, especially in the resources section showing resources inserted as spans. These are the rest of them...

Links

Markua's hyperlink support is a subset of that of Markdown. The **inline** link syntax is supported, as is the **automatic link** shortcut.

Inline Links

The normal way to create a link is as follows:

```
[link text](absolute_url)
```

Example:

```
Markua was developed at [Leanpub](http://leanpub.com).
```

Automatic Links

To create a link where the text displayed for the link text is the URL itself, the automatic link syntax can be used. In this syntax, an absolute URL is enclosed in angle brackets.

```
Some text <absolute_url> some text.
```

Example:

```
Markua was developed at <http://leanpub.com>.
```

Explicitly Creating Spans with `[...]`

Surrounding text in square brackets can be useful not just for giving it a URL to link to. If you wish to add attributes to an arbitrary span of text, you can create an arbitrary span of text using square brackets and then add an attribute list immediately afterward:

```
Some text [then a span]{and: an, attribute: list} attached to the span.
```

You can use any attribute list on this span, and you can also just use the id syntactic sugar `{#theid}` on this span. The most common uses of this are to add `id`s or index entries, which are discussed later.

Note, however, that you cannot start a normal span with a caret (^): this creates a `[^footnote]` instead. (Footnotes are discussed below.)

Sometimes a Square Bracket is Just a Square Bracket

If there are no round brackets or curly braces immediately after some text in square brackets, the text in square brackets is just that: text in square brackets. In this case, the square brackets are output as normal text.

This is useful when you want to [sic.] something, etc.

Footnotes and Endnotes

Footnotes

Books often have footnotes.

To add a footnote, you insert a footnote tag using square brackets, a caret and the tag, like this:

```
This has a footnote[^thenote].
```

Then, you define the footnote later in the document, using the same square brackets, caret and tag, followed by a colon, a space and the footnote definition:

```
[^thenote]: This is the footnote content.
```

If you wish to write multiple paragraphs in the footnote, you must indent the subsequent paragraphs by four spaces or one tab:

```
This has a footnote[^thenote].

Here is some unrelated text.

[^thenote]: This is the first paragraph of footnote content.

    This is the second paragraph of footnote content.

Here is some more unrelated text.
```

Whether the numbering of footnotes restarts every chapter is something that can be specified by the `restart-footnote-numbering` setting.

Endnotes

Books often have endnotes. Sometimes these are instead of footnotes, but other times, these are in addition to footnotes. So, it makes sense for Markua to define separate syntaxes for both, rather than just defining one "footnote or endnote" syntax and letting the author pick whether the notes are footnotes or endnotes via a global setting.

To add an endnote, you insert an endnote tag using square brackets, two carets and the tag, like this:

```
This has an endnote[^^thenote].
```

Endnotes are like footnotes, but happier (^^).

Then, you define the endnote later in the document, using the same square brackets, two carets and tag, followed by a colon, a space and the endnote definition:

```
[^^thenote]: This is the endnote content.
```

Just as with footnotes, if you wish to write multiple paragraphs in an endnote, you must indent the subsequent paragraphs by four spaces or one tab.

Whether the numbering of endnotes restarts every chapter is something that can be specified by the `restart-endnote-numbering` setting.

Single Reference to Footnotes and Endnotes

You can only refer to a footnote or endnote once. You can't define a footnote or endnote in one place and refer to it multiple times in the same Markua manuscript. If you wish to refer to a parenthetical piece of text from multiple places in a Markua manuscript, the best approach is to put it in a section (or sub-section, sub-sub-section, etc.) or aside and refer to it from multiple places using a crosslink.

Footnotes and Endnotes Support Required for Paragraphs Only

Leanpub only supports footnote and endnote references inserted in normal paragraph content.

(Sometimes authors want to get creative with their footnotes and endnotes. Sometimes they want to add them in headings, or in footnotes or endnotes themselves. This latter style has been used on rare occasions, most notably by David Foster Wallace. However, that's difficult to implement, and is optional in Markua, so Leanpub has not done so.)

Crosslinks and ids

There are two parts to making a crosslink.

1. Define an id.
2. Reference that id with a crosslink.

Defining an id

There are two ways to define an id:

1. Using an id attribute `{id: some-id}`
2. Using a shorter "syntactic sugar" approach: `{#some-id}`

The shorter "syntactic sugar" approach is usually preferred. However, it can look a bit odd in an attribute list with other attributes in it. So, in that case, the `{id: some-id}` syntax is preferred.

The `{#some-id}` syntax does not currently work when there are other attributes in the attribute list. So, in that case, you must currently use the `{id: some-id}` syntax if you are adding other attributes in a larger attribute list.

In terms of the value of an `id`, it has some special restrictions:

1. The first character in the `id` has to be a lowercase or uppercase letter, i.e. `[a-zA-Z]` if you think in regular expressions.
2. The remaining characters in the `id` have to be a lowercase or uppercase letter or a digit or a hyphen (-) or an underscore (_).
3. You can only define an `id` value once in an entire Markua document, even one that is split over multiple files.

These restrictions ensure that your `id`s can then be linked to by a crosslink from anywhere in the Markua document.

The id needs to be defined on either a block or span element.

> If an id is defined with an invalid name, Leanpub must ignore it and should provide a warning in the list of warnings it produces when generating the document.

Defining an id on a Block Element

To define an id on a block element like a paragraph, figure, heading or even a definition list item, you simply stick the id definition on a line above the start of the block element. Note that exactly one newline must separate the attribute list from the block element–if two newlines are used, the attribute list will be interpreted as a directive, and the id won't be correctly applied.

Here's how to use the attribute list syntax to define an id attribute:

```
{id: some-id}
This is a paragraph with the id of `some-id`.
```

Here's how to use the shorter "syntactic sugar" approach to define an id attribute:

```
{#some-id}
This is a paragraph with the id of `some-id`.
```

Defining an id on a Span Element

To define an id on a span element you simply add the id definition immediately after the span element.

Here's how to use the attribute list syntax to define an id attribute on a span element:

```
The word Markua{id: markua} has an id.

Leanpub is based in **Victoria, BC, Canada**{id: victoria}.
```

Here's how to use the shorter "syntactic sugar" approach to define an id attribute:

```
The word Markua{#markua} has an id.

Leanpub is based in **Victoria, BC, Canada**{#victoria}.
```

Here's how to define an id on a custom span:

```
The [quick sly fox]{#quick_sly} jumped over the lazy dogs.
```

If you want to define an id on a span while also defining other attributes like index entries, the `id:` syntax must be used in a full attribute list:

```
The [quick sly fox]{id: quick_sly, i: "Fox, Sly and Quick"} jumped over the lazy dogs.
```

Referencing an `id` With a Crosslink

Regardless of how you defined the id, you then link to it to create a crosslink. To do this, you use the `#` character and the id in a link:

```
[link text](#some-id)
```

This syntax is intended to be reminiscent of HTML anchor tags, not of `h1` titles in Markua.

Note that order of definition and use does not matter: crosslinks will work regardless of whether the `id` is defined before or after the use of it.

Rules for `id`s and Crosslinks

- If a Markua document contains duplicate `id` attribute values, the **first** one is used and the subsequent ones are ignored. Leanpub should output a warning about duplicate `id`s.
- Crosslinks that reference an unused `id` may either be created as a (broken, non-functional) link or be created as normal text (not a link) by Leanpub. The Leanpub may also output a warning about this somewhere, but not in the actual document text itself.

Character Substitution (`X--` for X—, `X --` for X –, `...` for …)

All Markua documents are written in UTF-8, so to produce any Unicode character, it possible to just use the proper Unicode characters. However, in certain cases, it's desirable for Markua to specify automatic replacement of certain combinations of characters with a Unicode replacement. If Leanpub encounters one of these combinations of characters outside of a code block, Leanpub must replace the combination of characters with the appropriate Unicode character in the output.

`--` To produce an em dash (—), what is thought of by non-typography people as a “dash” or a “long dash”, you can just type two hyphens (`--`) directly after a non-space character. You can also use the proper Unicode character, U+2014, of course. The following all produce em dashes: `foo--bar`, `foo-- bar`, `foo--`.

`--` To produce a space followed by an en dash (–), or the kind of dash that’s wider than a hyphen but narrower than an em dash, you can just type a space, followed by two hyphens (`--`). You can also use the proper Unicode character, U+2013, of course. The following both produce en dashes preceded by spaces: `foo -- bar`, `foo --`. (With `foo -- bar`, there’s a space before and after the en dash; with `foo --`, there’s no space after it (e.g. at the end of a paragraph).

`...` To produce a horizontal ellipsis (…), you can just type `...`. You can also use the proper Unicode character, U+2026, of course.

Optional Automatic Curly Quotes Outside of Code Blocks and Spans

Leanpub may replace the `"` character with the appropriate “curly quote” at its discretion. This lets `"typography"` become `“typography”`, and `it's` become `it’s` as appropriate.

Note that this is an optional behaviour: Leanpub may support this fully, only in some output formats, or not at all.

Also, note that it is **NEVER** acceptable for Leanpub to do this, or *any* character substitution, to text inside a code block or code span. In almost all instances this would completely break the code. (If you wonder how I got curly quotes into the code spans for `“typography”` and `it’s` above, it’s because I pasted them into the manuscript that way. Just as Leanpub doesn’t make straight quotes curly in a code span, it doesn’t make curly quotes straight in a code span either.)

Escaping Special Characters with Backslash (`\`)

Curly braces and backticks are special in Markua.

At the beginning of a line, an opening curly brace (`{`) starts an attribute list, and two opening curly braces (`{{`) start a placeholder. In the middle of a block element, an opening curly brace starts an index entry. And, at the top of a manuscript.txt file (if the single file approach is used), an opening curly brace starts a settings block.

So, to use a curly brace as an actual curly brace character, you need to backslash-escape it like this: `\{`. (Note that this does not apply inside code or other resources: Markua does not process anything inside them.)

Similarly, a backtick is special. In text content (such as this paragraph), a backtick starts an inline span resource such as a code resource. And three backticks on a line by themselves start a code block.

Code Spans and Backticks (`)

You can create a code span by using pairs of 1, 2 or 3 backticks to surround a span of text within a paragraph, like this:

```
This paragraph has a Ruby `puts "hello"` code span inside it.
```

You cannot show a literal backtick in a normal code span, however, since you cannot backslash escape anything inside it. Inside a code span, a backslash is just a backslash.

```
This paragraph has a Ruby `puts "hello\tworld"` code span inside it.
```

So, if you want to output a backtick in a code span, you need to delimit the code span using two backticks:

```
This paragraph has a code span with literal backticks `` `say hello` `` in it.
```

Metadata

Attributes

Attributes are used to do everything from specify the language of code blocks, add ids for crosslinking and even support extensions to Markua. We have already seen attributes in the attribute lists we have encountered.

Attribute List Format

An attribute list is one or more key-value, comma-separated pairs:

```
{key_one: value1, key_two: value_two, key_three: "value three!", key_four: true, key_five: 0, key_six: 3.14}
```

Note that you can skip the space between the colon and the value: the following `{format: ruby}` and `{format:ruby}` both work. However, for consistency I recommend always adding a space.

An attribute list can be inserted into a Markua document in one of three ways:

1. Immediately above a block element (e.g. heading, figure, aside, blurb, quiz, etc.), with **one newline** separating it from the block element.
2. Immediately after a span element (e.g. a word, italicized phrase, etc.) in normal paragraphs and in similarly-simple contexts, with **no spaces** separating it from the span element.
3. On a line by itself, with one blank line above and below it. In this format, the attribute list contains directives.

Regarding #2 and #3: Any line outside of a code resource which starts with an opening curly brace `{` and ends with a closing curly brace `}` is assumed to be an attribute list, and will not be output by Leanpub. If you want to start a line with a literal opening curly brace `{` you need to preface it with a backslash (`\`).

Regarding #2: There is some deliberate ambiguity here. If a Leanpub cannot handle the attribute list (including just an id attribute added with in the given context, it should just ignore the attribute list and add a warning to any list of warnings when generating the book. For example, Leanpub does not support attribute lists in headers or in captions. So, in Leanpub, you cannot do this:

```
# Chapter One{#one}

That doesn't work.

# Chapter{id: chapter-two} Two

That doesn't work either.

You also cannot say ![here's a horse{#horse} image](images/horse.jpg) in Leanpub.
```

In both cases, what you want to do is just define the attribute list on the outer element:

```
{#one}
# Chapter One

That works.

{id: chapter-two}
# Chapter Two

That works too.

You also can say ![here's a horse image](images/horse.jpg){#horse} in Leanpub.
```

Attribute Keys

The keys of attributes must consist exclusively of lowercase letters, hyphens (-) and underscores (_). Uppercase letters are not permitted in attribute keys: Leanpub must treat uppercase letters in attribute keys as an error.

If a key is duplicated in an attribute list, the first key value is used and subsequent ones are ignored. Leanpub should add a warning in its list of warnings, which are *not* output in the output itself.

Attribute Values

All attributes are text. Leanpub interprets text values of “true” and “false” as representing true and false. Quotes are optional for attribute values, and are only needed if the attribute value contains whitespace or special characters.

If a text attribute value contains a quote, it must be “escaped” with a backslash: e.g. `{caption: "\"Fresh\" Fish"}`

`id` Attributes

As previously discussed, there is special syntactic sugar for ids: `{#foo}` is equivalent to `{id: foo}`. However, ids are just attributes.

`title` Attributes

Markua headings (part, chapter, section, sub-section, etc.) and figures can all have `title` attributes specified in an attribute list. This is text which overrides what is displayed for the heading or figure in the table of contents. For a heading, it is analogous to the `title` attribute on a resource inserted as a figure, which specifies the text to use for the figure in the appropriate list of figures (e.g. List of Illustrations, Table of Tables, etc.). If a heading does not have a title attribute, the text of the heading itself is used–which is quite often exactly what is desired. Use of a `title` attribute is always optional; it's only used when the default behaviour of using the heading text (or the `caption` attribute for a resource) is not appropriate, say if it's too long.

Conditional Inclusion Attributes on Headings: `book`, `sample`

Markua headings (and *only* headings) may have various attributes which specify which output formats their content (of the part, chapter, section, sub-section, etc.) should be included in. If the given attribute is not present, the default value of it is that specified by the nearest ancestor heading. If no such attribute is present at a top-level heading, the default is given by the default value for the attribute defined of Markua.

There are eight attributes which specify conditional inclusion in output targets; four for books, and four for courses.

`book`
: If `true`, include this heading and its content (including nested sections, subsections, etc.) in the book (PDF, EPUB and/or MOBI) that is being generated. If `false`, omit it. The default is `true`. Setting this to `false` is an easy way to "comment out" a section of your book.

`sample`
: If `true`, include the content under this heading (including nested sections, subsections, etc.) in the sample of the book that is being generated. If `false`, omit it. The default is `false`. Note that this attribute just governs the inclusion of the content, not the heading itself. Leanpub may choose to include every heading in the sample version of a book or course, in order to produce a representative Table of Contents. In a case such as this where `sample` is `false`, Leanpub may output special content inside the chapter, section or subsection to indicate that the content itself is being omitted from the sample. This attribute applies to both the book version (PDF, EPUB and/or MOBI) and the web version of the sample.

This set of defaults means that by default, there is no book or course content included in the sample version of a book or course. As discussed, the headings themselves may all be included, in order to generate an appropriate Table of Contents for the sample book or course.

The reason that there are different attributes used for books and courses, instead of a course just reusing the same values that are used by a book, is that this way the same manuscript can be used to produce a book and a course. This approach is obviously not for everyone, but it may appeal to highly technical authors who wish to not repeat themselves, and who don't want to have to maintain separate git branches for a book and a course. Our assumption, however, is that most books or courses will require separate git branches, and that thus this approach merely saves some labour at the cost of some added complexity.

Note that specifying one of these attributes in a nested section overrides any value inherited from its ancestors, or from the default. This way, you can include a chapter in the sample, except for a specific section of the chapter.

Example:

```
{sample: true}
# Chapter One

This is included in the sample.

## Section One

This is included in the sample.

{sample: false}
## Section Two

This is *not* included in the sample.

## Section Three

This is included in the sample.

{book: false, sample: true}
# Buy the book!

What you read was just a sample. Why not buy the full book?

# Chapter Two

This is not included in the sample.
```

To be clear: ALL conditional inclusion attributes *ONLY* have meaning when used as an attribute list on headings, i.e. the things that are defined with 1-6 `#` signs followed by a space, followed by text.

For example, you can only say `{sample: true}` immediately above a heading. You can't have a blank line below it (otherwise it would be a directive, and not be valid) and you can't attach it to anything other than a heading (like a paragraph, figure, etc.).

Leanpub must raise an error if it encounters a conditional inclusion attribute used incorrectly, so as to help the author understand how to properly use them.

Directives

Directives are switches which affect the future behaviour of Leanpub.

The syntax for directives is simple: they are just contained in an attribute list. The only difference is that the attribute list is inserted an a line by itself, with one blank line above and below it. (There are two exceptions: if the directive is at the beginning of a manuscript file, you can omit the blank line above it; if it's at the end of a manuscript file, you can omit the blank line below it.)

A directive does not have any kind of "closing tag"–it simply remains in effect for the rest of the Markua document, or until the directive is overridden by another use of the same directive with a different value.

Currently the only directives supported by Leanpub are:

- `{frontmatter}` (for Roman numeral numbering of things like a preface)
- `{mainmatter}` (for the start of normal numbering)
- `{backmatter}` (to indicate the start of back matter, like appendices)

Here's an example:

```
{frontmatter}

# Preface

Foo

{mainmatter}

# Chapter One

Bar

# Chapter Two

Baz

{backmatter}

# Appendix

Hello world.
```

These directives have not yet been updated to `section` directives per the Markua Spec.

Appendices

No Inline HTML

Inline HTML, in which HTML is inserted directly in the content of a Markua document, is not supported in Markua.

HTML is just one possible output format, and other possible Markua output formats (such as PDF) are not based on HTML. If inline HTML was supported, Leanpub would have to support parsing and meaningfully outputting all of HTML syntax as well as all of Markua syntax.

Markua and Markdown have different use cases. Markdown is a better way to write HTML; Markua is a better way to write a book. Since Markdown's only output format target is HTML, it might as well support inline HTML: generating HTML from HTML is as simple as passing the HTML through. From an implementation perspective, Markdown gets inline HTML support for free.

By not supporting inline HTML, Markua imposes more constraints on writers who would be tempted to use inline HTML for layout purposes. Since Markua does not support inline HTML, attempting to do complex layout in Markua using HTML is just not possible. And since it's not possible, the temptation to procrastinate by doing formatting is reduced.

Now, one design benefit of the support for inline HTML in Markdown is that Markdown's syntax can stay artificially small–since Markdown authors can always fall back to using HTML directly, Markdown does not need to be able to produce all of HTML. Since Markua does not support inline HTML, Markua must contain all concepts that it supports directly expressed using Markua syntax. For example, there is no official table syntax in Markdown, and Markdown authors can just use inline HTML tables. Since Markua does not support inline HTML, and since books often require tables, Markua needs to add a table syntax.

Differences with Markdown

Markua has a number of differences from Markdown as defined by John Gruber.

These are the main ones:

1. In Markdown, `*one asterisk*` and `_one underscore_` both produce *italics*. In Markua, `*one asterisk*` produces *italics*, and `_one underscore_` produces an ***underline*** unless the `italicize-underlines` global setting is set to true.
2. Markdown supports inline HTML; as discussed earlier, Markua does not.

3. Markua defines many more types of numbered list numbering than Markdown. In Markdown, the only type of numbering supported is decimal numbering starting from 1. If you need any more features, you need to use inline HTML. However, since Markua does not support inline HTML, Markua provides more list types and features.
4. Unlike in Markdown, in Markua the number that begins the list in the manuscript is the number that begins the list in the output.
5. Markdown does not specify a table syntax. Since Markdown supports inline HTML, it does not need to–if you want a table, you can simply use an inline HTML table. Since Markua does not support inline HTML, Markua uses the GFM table syntax.
6. Markua supports definition lists; Markdown does not (except via inline HTML).
7. Markdown lets you use a plus sign (`+`) before each item in an unordered ("bulleted" in Markua) list; Markua does not. Having three syntaxes is just overkill, so Markua drops the plus–it's far less common than the asterisk and hyphen. This is the same decision made by GitHub Flavored Markdown, which also supports[14] `*` and `-` but not `+`.
8. Markdown does not support closing parentheses (`)`) as ordered ("numbered" in Markua) list delimiters; Markua does.
9. Markua and Markdown currently handle spaces, newlines and indentation differently. This is discussed in the Whitespace: Spaces, Tabs and Newlines section.
10. Markdown's reference link syntax and its implicit link name shortcuts are currently not supported in Markua.
11. Markua's heading syntax is currently a subset of Markua's atx headers.
12. The backtick syntax of adding inline code resources as spans is identical to the code span syntax of Markdown, however the optional attribute list is Markua-specific.
13. The syntax for inline code resource figures is similar to the "fenced code blocks" syntax of many Markdown extensions, such as PHP Markdown Extra and GitHub Flavoured Markdown.
14. The "four space indent" method of creating code blocks in Markdown is currently not supported in Markua.
15. In Markdown links, a URL can be either absolute (starting with `http://` or `https://`) or relative (just referring to a path like /foo/bar.html), since relative URLs can make sense when writing blog posts which live on web servers. In Markua, however, all URLs must be absolute.

[14] https://help.github.com/articles/markdown-basics/