Leonardo Giordani

# The Mau Book

3nd Edition - 2023

# Contents

# Introduction

A beginning is the time for taking the most delicate care that the balances
are correct.
*Frank Herbert, Dune (1965)*

Mau is a lightweight markup language heavily inspired by Markdown and AsciiDoc that makes it a breeze to write blog posts or books. If you already know Markdown or AsciiDoc you already know which type of software Mau is, and you will quickly learn its syntax.

The main goal of Mau, however, is to provide a *customisable* markup language. While Mau's syntax is fixed by its implementation, its output is created through user-provided templates. This strategy gives the user great flexibility with no added complexity.

I currently use Mau to write posts for my blog The Digital Cat and to write books like Clean Architectures in Python. This documentation is also written using Mau. So, as you can see, the system is production-ready, and you can start using it today.

## The story so far

Markdown is a great format, and I used it for all the posts in my blog since I started writing. Pelican, which is the static site generator that I use, supports Markdown out of the box, so it was extremely easy to start using it, and overall I had an enjoyable experience.

When the idea of a book about the clean architecture began to take shape in my mind, a quick survey of the platforms for self-publishing led me to LeanPub, which provides a good toolchain based on their Markdown dialect called Markua. Being so similar to Markdown, the transition was seamless for me, and I could publish the first edition of the book without any issues.

In the meanwhile, my activity on the blog increased, and I started to feel the need to add features to my articles that weren't easily created with Markdown, such as adding a file name and callouts to the code blocks or adding admonitions. Sure, such things can be added using raw HTML, but that popped the bubble of the simple markup syntax, so I wasn't happy with that solution.

The same problems arose when I started working on the second version of the book, with some additional concerns. Since the book is freely available, I wanted to use the same source code to generate a website and be able to reuse the same features both in the resulting HTML and in the PDF.

I couldn't find a good way to create tips and warnings using Markdown. Recently, Python Markdown added a feature that allows specifying the file name for the source code, but the resulting HTML cannot easily be changed, making it difficult to achieve the graphical output I wanted through CSS. So, I started looking into other projects.

I tried Pandoc, and a week spent trying to learn again that black magic called TeX was enough for me to decide that the system wasn't what I needed. My relationship with TeX/LaTeX has always been stormy: while I admire the system, the ingenuity, and the one-man show effort behind TeX, the final result is a convoluted beast that is difficult to tame. It is also terribly undocumented!

The third system that I found was AsciiDoc, which started as a Python project, abandoned for a while and eventually resurrected by Dan Allen with Asciidoctor. AsciiDoc has a lot of features and I consider it superior to Markdown, but Asciidoctor is a Ruby program, and this made it difficult for me to use it. In addition, the standard output of Asciidoctor is a nice single HTML page but again customising it is a pain. I eventually created the site of the book using it, but adding my Google Analytics code and a `sitemap.xml` to the HTML wasn't trivial, not to mention customising the look of elements such as admonitions.

In the end, I wasn't completely happy with Asciidoctor, and once again I started looking around to see if there was something that matched my requirements.

## What I was looking for

In a nutshell, this is what I was hoping to find:

- A simple markup syntax [Markdown, Markua, Asciidoctor]
- A stand-alone implementation that I can run locally [Markdown, Asciidoctor]
- A Python implementation that can be used from Pelican [Markdown]
- Support for admonitions and callouts [Asciidoctor]
- PDF output [Asciidoctor]
- Highly configurable HTML output []

As you can see none of the systems could tick all the boxes, and all of them are missing a way to easily change the output of the rendering.

### What I did

Since no existing tool was matching my requirements I did what people like me do when they lack a tool. I wrote it myself!

I have been studying compilers all my life, even though I can by no means be called an expert. I have a series of posts on my blog where I write an interpreter in Python, based on the amazing work of Ruslan Spivak, so I thought that I might have at least tried to create a Python interpreter for Asciidoctor's syntax since the original AsciiDoc code was left unmaintained (apparently development started again later).

After one month I had a working tool that I successfully connected with Pelican and used to render some posts that I had already written in Markdown. I don't consider the project revolutionary, but I can honestly say that the day I saw Mau working for the first time is one of the best days of my career as a software developer. At that point, Mau had already slightly diverged from the original idea, though.

While initially I was aiming to an implementation of AsciiDoctor's syntax, and retained a great deal of it, I took the opportunity to try a different path when it came to rendering. Having already successfully used Jinja2 in other contexts, I had this idea of using Jinja templates to render Mau's output, so that the user could either use the standard one or provide their own and thus easily customise the final result.

I later wrote a visitor (a rendering class) that converts Mau's input into AsciiDoctor or Markua, and even though it doesn't cover all the features of the two languages, it allowed me to use Mau to rewrite my book and publish it online while using the Markua output to feed Leanpub's processing chain that produces the PDF.

## Where we are now

The short story is that Mau works, and as I already mentioned is used for both my blog and my books. Mau's features are

- A simple markup syntax
- A stand-alone implementation that you can run locally on any system that supports Python3
- A plugin for Pelican that allows you to use Mau to write blog posts and website pages
- Full support for a good range of standard HTML features (paragraphs, lists, headers, ...) and for some advanced ones such as admonitions, code callouts, includes, and footnotes.
- Extremely configurable output using Jinja2 templates

- Stand-alone PDF creation (since version 3)

I learned a lot writing Mau, and I'm happy that the whole idea proved worth the time I invested. I'd love to know that other people found it useful, so in this manual I will show you how to install and use Mau for your projects.

Thanks for giving Mau a try!

## How to contribute

Feel free to send PRs to fix this book should you find typos or any other mistake. Contributions are more than welcome!

# Install and run Mau

"I'll tell you. I'm going to install a visiplate right over my desk. Right on the wall over there, see!"
*Isaac Asimov, "I, Robot" (1950)*

In this chapter I will show you how to install Mau both as a stand-alone tool and as a plugin for Pelican. The only requirements are a working Python3 installation and a virtual environment.

While Python comes preinstalled in several operating systems these days, you can find detailed instructions on how to install it on the Python official website. Virtual environments are covered in the official documentation, but you might want to also explore tools to manage them, like pyenv etc. These are however not mandatory to run Mau.

## Install Mau as a stand-alone tool

Mau is available on PyPI and to install it you just need to run

```
pip install mau
```

At this point you should have the command line tool `mau` available. You can test it with

```
mau --version
```

You can get help directly on the command line running

```
mau --help
```

You will notice in the help text that the option `-f` is used to specify the output format, and that the only available output is `dump`, which prints out the Abstract Syntax Tree. To render Mau source

code you need a visitor plugin, and at the moment the available ones are the HTML visitor and the TeXvisitor. You can install only one of them or both.

```
pip install mau-html-visitor
pip install mau-tex-visitor
```

Now, if you run `mau -help` you will see the new output format available to the option `-f`.

## Run Mau as a stand-alone tool

The simplest command line for Mau is

```
mau -f FORMAT -i INPUT
```

that reads the Mau source file `INPUT` and converts it into the desired format saving the output in a file called `INPUT.${ext}`, where `${ext}` is a suitable extension that depends on the chosen format. If the input file has the extension `.mau` that will be automatically removed.

If you want to specify the output format and the name of the output file you can use the two options `-f` and `-o`

```
mau -f FORMAT -i INPUT -o OUTPUT
```

where `FORMAT` is one of the output formats that Mau supports, and both `INPUT` and `OUTPUT` are full paths, extensions included.

### Example

Make sure you installed the HTML visitor and create the file `test.mau` in the current directory with this content

```
= A test

This is a test for the Mau markup processor.
```

Now you can run

```
mau -f html -i test.mau -o test.html
```

That will parse the content of `test.mau` and render it as HTML into `test.html`. The content of that file is

```
<html>
  <head>
  </head>
  <body>
    <h1 id="a-test">A test</h1>
    <p>This is a test for the Mau markup processor.</p>
  </body>
</html>
```

(minus the formatting which was added here for clarity)

You can now open the output file with your browser

```
firefox test.html
```

and enjoy your first document created with Mau.

### Configuration file

Mau supports a configuration file written in YAML that can be loaded with the option `-c`

```
mau -c config.yml -f html -i test.mau -o test.html
```

Each value defined in the config file is stored as a variable under the namespace `mau`, and can be used in the Mau source. See the section about variables to know more about this.

## Using Mau programmatically

You can use Mau programmatically in your Python code. You fist need to import `Mau` and the function to load the visitor plugins

```
from mau import Mau, load_visitors
```

The object `Mau` has the following constructor

```
class Mau:
    def __init__(
        self,
        input_file_name,
        visitor_class,
        config=None,
        custom_templates=None,
        templates_directory=None,
        full_document=False,
    ):
```

- `input_file_name` is used by Mau when a syntax error is found. Mau tries to point out where the syntax error is and it includes the name of the file.
- `visitor_class` is the Python class that implements the conversion into the target format.
- `config` is a dictionary containing the configuration.
- `custom_templates` is a dictionary containing custom templates.
- `templates_directory` is a directory that contains a file for each template you want to customise.
- `full_document` tells Mau if the output is a fragment or if it has to be wrapped in a document using the specific template (which might add header and footer, for example).

To initialise it you need to load the visitor plugins and to select the one corresponding to the output format you want.

```
visitor_classes = load_visitors()
visitor_class = visitors["html"]
```

Then you might customise the configuration and the templates

```
config = {
    "some_var": 42
}

custom_templates = {
    "header.html": '<h{{ level }} id="{{ anchor }}">{{ value }}</h{{
    ↪  level }}>',
}
templates_directory = "/some/templates/dir"
```

As you can see from the prototype, all these values are optional. At this point you can initialise the object `Mau`

```
mau = Mau(
    "/some/source/path",
    visitor_class=visitor_class,
    config=config,
    custom_templates=custom_templates,
    templates_directory=templates_directory,
)
```

You can then call the lexer, the parser, and the visitor on some Mau text contained in a string

```
text = "Some Mau text contained in a string"
lexer = self._mau.run_lexer(text)
parser = self._mau.run_parser(lexer.tokens)
content = self._mau.process(parser.nodes, parser.environment)
```

Last, some visitors contain a function to transform the output, for example to tidy it up, so you might want to call it

```
if visitor_class.transform:
   content = visitor_class.transform(content)
```

The code of the whole example is then

```
from mau import Mau, load_visitors

visitor_classes = load_visitors()
visitor_class = visitors["html"]

config = {
    "some_var": 42
}

custom_templates = {
    "header.html": '<h{{ level }} id="{{ anchor }}">{{ value }}</h{{
    ↪  level }}>',
}
templates_directory = "/some/templates/dir"

mau = Mau(
    "/some/source/path",
    visitor_class=visitor_class,
    config=config,
    custom_templates=custom_templates,
    templates_directory=templates_directory,
)

text = "Some Mau text contained in a string"
lexer = self._mau.run_lexer(text)
parser = self._mau.run_parser(lexer.tokens)
content = self._mau.process(parser.nodes, parser.environment)

if visitor_class.transform:
   content = visitor_class.transform(content)
```

## Using Mau in Pelican

You can use Mau to write posts and pages in Pelican. First you need to install the plugin that enables it

```
pip install pelican-mau-reader
```

You can see the updated documentation about the plugin on the project page but overall you just need to follow the instructions in this paragraph.

The basic usage of the plugin is simple. Every file in your content directory that ends with `.mau` will be processed by it, and you need to specify metadata using Mau's variables under the namespace `pelican`. For example

```
:pelican.title:This is a post written with Mau
:pelican.date:2021-02-17 13:00:00
:pelican.modified:2021-02-17 14:00:00
:pelican.category:tests
:pelican.tags:foo, bar, foobar
:pelican.summary:I have a lot to write
```

The syntax `:name:value` is used by Mau to create variables and you can learn more about it in the dedicated chapter.

# Chapter 1

# Basic syntax

Some of the newer ones are having trouble because they never really
mastered some basic techniques, but they're working hard and improving.
*Orson Scott Card, "Ender's Game" (1985)*

This and the following chapters will give you an overview of the basic syntax for paragraphs, styles, and inline elements like links and images. If you are familiar with either Markdown or Asciidoctor none of these will be a surprise to you, but I will describe everything assuming the reader doesn't know any markup language.

I will assume you followed the instructions in the previous chapter and know how to run Mau in a stand-alone fashion, or that you are running it in Pelican. Either way, I will refer to generic input and output, meaning respectively the Mau source and the final result. Unless explicitly stated, the rest of the book will use the HTML output format.

I will show the source code in a block like this

Mau source

```
This is Mau source code
```

If useful, the resulting HTML code will be shown in a block like this

HTML output

```
<p>The resulting HTML code</p>
```

As this book is formatted using LaTeX, the rendered version won't use the HTML code but its LaTeX counterpart. The rendered version will be shown with an aside like

> This is the rendered output

When showing the HTML output I will skip the boilerplate to keep examples compact and readable. When you see a block with the resulting HTML like

HTML output

```
<p>The resulting HTML code</p>
```

the full output is actually

HTML output

```
<html>
  <head>
  </head>
  <body>
    <p>The resulting HTML code</p>
  </body>
</html>
```

Finally, remember that both Markdown and Asciidoctor have a fixed rendering, while the output in Mau is ruled by templates. What you see in the output HTML and in the rendered boxes is the result of Mau's default templates, but those can be overridden at any time, as we will see in a later chapter.

At the end of this book you will find a description of the attributes each element available in Mau exposes to the template engine.

## Paragraphs

The simplest element in Mau is a line of text that forms a paragraph

Mau source

```
This is a line of text.
```

This is a line of text.

Adjacent lines of text are automatically joined into a single paragraph, so both

Mau source

```
This is a sentence. And another sentence.
```

and

Mau source

```
This is a sentence.
And another sentence.
```

will both be rendered as

This is a sentence. And another sentence.

To separate paragraphs you need to insert one or more empty lines. For example

Mau source

```
This is a sentence.

And another sentence.
```

and

Mau source

```
This is a sentence.



And another sentence.
```

will both be rendered as

This is a sentence.

And another sentence.

## Comments

Mau supports both single-line and multi-line comments. A single line comment uses double slashes

Mau source

```
This is a sentence.

And another sentence.

// This is a comment and won't be rendered
```

This is a sentence.

And another sentence.

Multi-line comments are surrounded by four slashes on a separate line

Mau source

```
1  This is a sentence.
2
3  And another sentence.
4
5  ////
6  This is also a comment
7  but it's spread on
8  multiple lines
9  for fun and to be a
10 little more readable
11 ////
```

This is a sentence.

And another sentence.

# Arguments

As we will see later, some advanced commands in Mau accept arguments, which are not too different from standard function arguments in programming languages. The most important thing to remember is that Mau values are always strings, so arguments do not really have types.

Arguments can appear between round brackets `()`, square brackets `[]`, or directly after a command, depending on the context. This will be clearly specified in each section when arguments are discussed.

Arguments can be **unnamed** or **named**. An unnamed argument has just a value, for example `html`, while a named one has a key and a value linked by an equal sign, like `language=html`. There can't be a space between the key and the value, so an argument like `language = html` is invalid.

Multiple arguments are separated by commas, optionally followed by one or more spaces. So, `source,html` and `source, html` are both valid, and the same is valid for named arguments, as `source,language=html` and `source, language=html` are equally accepted by the parser.

In the documentation, arguments will be given uppercase (e.g. `TYPE`), and named ones will followed by an equal sign (e.g. `LANGUAGE=`). You can avoid specifying the key if you pass argument values respecting the order provided in the documentation. So, if in a certain context you have to provide `TYPE, LANGUAGE=` you can either give `source, html` or `source, language=html`. Unnamed arguments can never follow named ones.

If an argument contains one or more spaces or commas you need to surround it with quotes, e.g. `attribution="J.R.R. Tolkien"` or `classes="class1,class2"`.

If the arguments are surrounded by braces you should use quotes also whenever the value contains the closing bracket. E.g. `(attribution="The (real) author")`. Here, the `)` after `real` would be considered the closing bracket if not surrounded by quotes.

If you need to pass quotes as value of an argument you need to escape them, e.g. `attribution="The so-called \"author\""`.

## Tags

Tags are special arguments prefixed by `#`. Whenever Mau finds an unnamed argument that starts with `#` it will strip the prefix and collect the argument in a separate list called `tags` (see the chapter about the node format). Tags are useful to categorise elements and apply special formatting.

For example the arguments list `source, language=html, #custom` contains the tag `custom` (plase note the missing `#`).

## Macros

Mau allows you to run functions called "macros". Macros always come in the form `[NAME](ARGUMENTS)` where `NAME` is the name of the macro and `ARGUMENTS` is a string formatted as described in the previous section. Built-in macros will be described in the following chapters.

## Include other files

So far Mau has a basic support for including other files through the directive `#include`. Directives are special commands that are processed before the Mau code is properly parsed

Mau source

```
::#include:/path/to/file.mau
```

Since the inclusion happens during the very first stages of the processing, you can also include files inside blocks (see the dedicated chapter).

# Chapter 2

# Text formatting

Leafing through the pages, he saw the book was printed in two colours.
There seemed to be no pictures, but there were large, beautiful capital letters
at the beginning of the chapters.
*Michael Ende, "The Neverending Story" (1979)*

## Basic styles

Inside paragraphs (and in other elements that support it) you can use text formatting which allows you to give text a certain style. Text formatting is done through symbols, enclosing a set of words between two identical symbols. Currently Mau supports stars (*), underscores (_), caret (^), tilde (~), and backticks (‘).

Mau source

```
Stars identify *strong* text.

Underscores for _emphasized_ text.

Carets for ^superscript^ and tildes for ~subscript~.

Backticks are used for `verbatim` text.
```

Stars identify **strong** text.

Underscores for *emphasized* text.

Carets for $^{\text{superscript}}$ and tildes for $_{\text{subscript}}$.

Backticks are used for `verbatim` text.

---

Text styles can be used together but backticks have a very strong behaviour in Mau

Mau source

```
You can have _*strong and empashized*_ text.

You can also apply styles to _*`verbatim`*_.

But verbatim will `_*preserve*_` them.
```

You can have ***strong and empashized*** text.

You can also apply styles to ***`verbatim`***.

But verbatim will `_*preserve*_` them.

---

Styles can be applied to only part of a word and do not need spaces

Mau source

```
*S*trategic *H*azard *I*ntervention *E*spionage *L*ogistics
↪  *D*irectorate

It is completely _counter_intuitive.

Parts of words can be ^super^script or ~sub~script.

There are too many `if`s in this function.
```

**S**trategic **H**azard **I**ntervention **E**spionage **L**ogistics **D**irectorate

It is completely *counter*intuitive.

Parts of words can be $^{\text{super}}$script or $_{\text{sub}}$script.

There are too many `ifs` in this function.

---

Using a single style marker doesn't trigger any effect. If you need to use two of them in the sentence, though, you have to escape at least one

Mau source

```
You can use _single *markers.

But you \_need\_ to escape pairs.

Even though you can escape \_just one_ of the two.

If you have \_more than two\_ it's better to just \_escape\_ all of
↪  them.

Oh, this is valid for `verbatim as well.
```

You can use _single *markers.

But you _need_ to escape pairs.

Even though you can escape _only one_ of the two.

If you have _more than two_ it's better to just _escape_ all of them.

Oh, this is valid for ‘verbatim as well.

## Text classes

You can assign specific classes to part of the text. The way classes are used and rendered depends on the output format. If the output is HTML, those will become CSS classes, while PDF output at the moment doesn't do anything specifc. To give text a class use the macro `class`

Mau source

```
This is [class]("text wrapped", myclass) in a class.
```

The HTML output of the code above is

HTML output

```
<p>This is <span class="myclass">text wrapped</span> in a class.</p>
```

---

You can specify multiple classes providing them in a comma-separated string

Mau source

```
This is [class]("text wrapped", "myclass1,myclass2") in multiple classes.
```

Resulting in

HTML output

```
<p>This is <span class="myclass1 myclass2">text wrapped</span> in
↪  multiple classes.</p>
```

Please note that the text passed to the macro `class` can contain Mau code such as styles.

Mau source

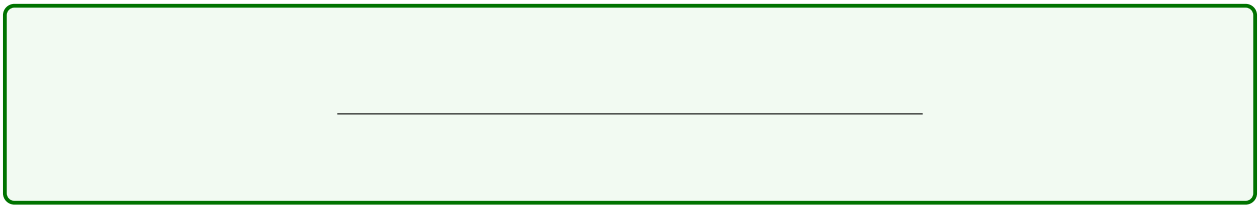
```
This is [class]("*text with styles* _wrapped_", "myclass") in a class.
```

## Horizontal line

You can add a separator or horizontal line using three dashes – on a separate line.

Mau source

```
---
```



# Chapter 3

# Links

> I had to nod. I was not unaware of the weakness of that link in my chain of speculations. Still, there were so many unknowns... I could offer alternatives, such as Random then did, but guesses prove nothing.
>
> *Roger Zelazny, "The Chronicles of Amber - Sign of the Unicorn" (1975)*

Since markup languages are mostly used to write hyperlinked documents, links play a big part in the syntax. Mau's implementation of links uses a syntax shared by all macros in the language (see the chapter about macros)

Mau source

```
The source code can be found
at [link](https://github.com/Project-Mau/mau).
```

The source code can be found at https://github.com/Project-Mau/mau.

The values between round brackets are arguments, as `link` is a macro.

## Add text to the link

You can add an optional text to the link that will replace the URL in the rendered text

Mau source

```
The source code can be found
on [link](https://github.com/Project-Mau/mau, GitHub).
```

The source code can be found on GitHub.

## Using spaces and quotes

If the title contains spaces or the closing parenthesis `)` you need to wrap it between double quotes

Mau source

```
The source code can be found
on [link](https://github.com/Project-Mau/mau, "the GitHub page").
```

The source code can be found on the GitHub page.

---

Should you need to add quotes to the title you will have to escape them

Mau source

```
The main [link](https://github.com/Project-Mau/mau, "\"repository\"").
```

The main "repository".

---

The same rules are valid for the URL

Mau source

```
A [link]("https://example.org/?q=[a b]","URL with special characters").
```

A URL with special characters.

## Mailto links

Mailto links can be created with the `mailto` macro, which works like the `link` macro described above.

Mau source

```
Get in touch at [mailto](info@example.com).
```

Get in touch at info@example.com.

# Chapter 4

# Headers

The abbot gave him a brief glare and began reading. The silence was awkward. “You found this over in the ‘Unclassified’ section, I believe?” he asked after a few seconds.
*Walter M. Miller Jr., "A Canticle for Leibowitz" (1959)*

To properly structure some text you need to divide it into sections and the best way to highlight sections is through headers. Mau supports them and automatically stores them in a table of context.

To create a header in Mau use the symbol = followed by the text of the header

Mau source

```
= A very important section
```

HTML output

```
<h1 id="a-very-important-section">A very important section</h1>
```

As you can see Mau converts it into a tag `h1` and automatically assigns an ID to it. The level of the header is ruled by the number of = symbols that you use. So, to create a header of level 3 you can write

Mau source

```
=== A less important section
```

HTML output

```
<h3 id="a-less-important-section">A less important section</h3>
```

## Headers and TOC

Mau stores all headers in a Table of Contents that can be created with the command `::toc:`

Mau source

```
= Main section

== Secondary section

== Another secondary section

=== A very specific section

::toc:
```

This will be rendered as

HTML output

```
<h1 id="main-section-06bc">Main section</h1>
<h2 id="secondary-section-3cbf">Secondary section</h2>
<h2 id="another-secondary-section-1dac">Another secondary section</h2>
<h3 id="a-very-specific-section-feca">A very specific section</h3>
<div>
  <ul>
    <li><a href="#main-section-06bc">Main section</a>
      <ul>
        <li><a href="#secondary-section-3cbf">Secondary
        ↪  section</a></li>
        <li><a href="#another-secondary-section-1dac">Another
        ↪  secondary section</a>
          <ul>
            <li><a href="#a-very-specific-section-feca">A very
            ↪  specific section</a></li>
          </ul>
        </li>
      </ul>
    </li>
  </ul>
</div>
```

You can avoid including a specific header in the TOC adding a tag and excluding it from the TOC using the argument `exclude_tag`. This will also exclude all children of that node.

Mau source

```
= Section 1

== Section 1.1

[#notoc]
== Section 1.2

=== Section 1.2.1

== Section 1.3

::toc:exclude_tag=notoc
```

This will exclude from the rendered TOC both `Section 1.2` and `Section 1.2.1`, but not `Section 1.3`.

## Anchors

Headers are automatically assigned an identifier by Mau, which is linked in the Table of Contents.

Mau source

```
= A very important section
```

HTML output

```
<h1 id="a-very-important-section">A very important section</h1>
```

The identifier is generated by an internal function that takes into account the text and the level of the header to avoid clashes. However, the ID is NOT granted to be unique. The function that generates the identifier can be replaced when using Mau programmatically, providing it in the variable `mau.header_anchor_function`.

For example you can do something like

```
def custom_header_anchor(text, level):
    return f"{text[:5]}-{level}"

config = {
    "header_anchor_function": custom_header_anchor
}

...

mau = Mau(
    "/some/source/path",
    visitor_class=visitor_class,
    config=config,
)
```

# Chapter 5

# Variables

It was wrong to have a third party present when I confronted you. It introduced one variable too many. It is a mistake that must be paid for, I suppose.

*Isaac Asimov, "Second Foundation" (1953)*

Mau supports variables of two types: strings and booleans. You can define variables at any point in a Mau file with the syntax `:NAME:VALUE` and then insert the value using the syntax `{NAME}`. For example

Mau source

```
:answer:42

The Answer to the Ultimate Question of Life,
the Universe, and Everything is {answer}.
```

The Answer to the Ultimate Question of Life, the Universe, and Everything is 42.

---

Variables can be used in several contexts. In paragraphs, headers, and footnotes they are useful to place constant strings that are repeated over and over and might need to be changed. Variables are replaced very early in the Mau translation process, so they can contain Mau code.

Mau source

```
:answer:*42*
:wikipedia_link:[link]("https://en.wikipedia.org/wiki/42_(number)")

The Answer to the Ultimate Question of Life,
the Universe, and Everything is {answer}.

You can learn more about it here {wikipedia_link}
```

The Answer to the Ultimate Question of Life, the Universe, and Everything is **42**.

You can learn more about it here https://en.wikipedia.org/wiki/42_(number)

Variables can also be used in block definitions, see the chapter about blocks for more information.

---

You can prevent variable replacement escaping the curly braces

Mau source

```
:answer:42

The Answer to the Ultimate Question of Life,
the Universe, and Everything is \{answer\}
```

The Answer to the Ultimate Question of Life, the Universe, and Everything is {answer}

---

As curly braces are used a lot in programming languages, Mau automatically escapes them when they are included in verbatim text

Mau source

```
:answer:42

The Answer to the Ultimate Question of Life,
the Universe, and Everything is `{answer}`
```

The Answer to the Ultimate Question of Life, the Universe, and Everything is `{answer}`

## Boolean variables

Variables without a value will automatically become booleans. The default value is true, which will become `True` when printed

```
:flag:

The flag is {flag}.
```

The flag is True.

You can create a false boolean variable negating it with an exclamation mark

Mau source

```
:!flag:

The flag is {flag}.
```

The flag is False.

When printed, boolean variables will be either `True` or `False`, which might not be very effective. They are however useful as flags in conditional blocks, see the chapter about blocks for more information.

## Namespaces

Variables can be created under a specific namespace using a dotted syntax

Mau source

```
:value:5
:module.value:6

The values are {value} and {module.value}.
```

The values are 5 and 6.

---

Mau's configuration values are available under the `mau` namespace (see the section about configuration for more information).

Namespaces are useful to communicate with external tools. For example, the Pelican metadata of this page are specified as

Mau source

```
:pelican.title:Variables
:pelican.slug:maubook-variables
:pelican.series:Mau - A template-based markup language
```

which allows the Mau reader plugin to extract them and pass them to Pelican.

# Chapter 6

# Lists and footnotes

- “Most people will tell you they know their weaknesses. When asked, they tell you, ‘Well, for one thing I’m overgenerous.’ Come on, then; list yours if you must. That’s what innkeepers are for.”

- “Well, for one thing I’m overgenerous, especially to innkeepers.”

*David Gemmell, "Legend" (1984)*

## Lists

Mau supports the creation of lists. You can create unordered lists using one or more characters `*` according to the level of the element

Mau source

```
* List item
** Nested list item
*** Nested list item
* List item
** Another nested list item
* List item
```

- List item
  - Nested list item
    - Nested list item
- List item
  - Another nested list item
- List item

and use the character # to create an ordered list

Mau source

```
# Step 1
# Step 2
## Step 2a
## Step 2b
# Step 3
```

1. Step 1
2. Step 2
   a) Step 2a
   b) Step 2b
3. Step 3

---

You can mix ordered and unordered lists

Mau source

```
* List item
** Nested list item
### Ordered item 1
### Ordered item 2
### Ordered item 3
* List item
```

- List item
  - Nested list item
    1. Ordered item 1
    2. Ordered item 2
    3. Ordered item 3
- List item

---

All spaces before or after the initial characters are ignored.

Mau source

```
* List item
  ** Nested list item
    *** Nested list item
*   List item
  **    Another nested list item (indented)
*   List item
```

- List item
  - Nested list item
    - Nested list item
- List item
  - Another nested list item (indented)
- List item

## Footnotes

You can insert a footnote including a definition and a reference. A definition contains the text of the footnote and is created with a block of type `footnote` (see chapter about blocks) and giving it a name

Mau source

```
[footnote, thename]
----
This is the text of the footnote.
----
```

Then you need to reference the footnote in a paragraph using the macro `footnote` and the name

Mau source

```
This is a paragraph with a note[footnote](thename).
```

Mau doesn't automatically include the definition of the footnote in the output. This has to be done with the command `::footnotes:`

```
This is a paragraph with a note[footnote](thename).

[footnote, thename]
----
This is the text of the footnote.
----

::footnotes:
```

This is a paragraph with a note[a].

[a] This is the text of the footnote.

# Chapter 7

# Images

Strange images appeared and vanished, flickering at the extreme limits of
visibility - vast faces, enormous hands, and things Garion could not name.
The turret itself trembled as the two dreadful old men ripped open the fabric
of reality itself to grasp weapons of imagination or delusion.
*David Eddings, "Magician's Gambit" (1983)*

You can include images in Mau documents, both as stand-alone elements and inline, mixed with text.

## Stand-alone images

Images can be included in the document with the command <<

Mau source

```
<< image:https://via.placeholder.com/150
```

---

You can add a caption to the image using a title

Mau source

```
. This is the caption
<< image:https://via.placeholder.com/150
```

HTML output

```
<div class="imageblock">
  <div class="content">
    <img src="https://via.placeholder.com/150"/>
    <div class="title">This is the caption</div>
  </div>
</div>
```

---

You can also specify the alternate text with the attribute `alt_text`

Mau source

```
[alt_text="Description of the image"]
<< image:https://via.placeholder.com/150
```

HTML output

```
<div class="imageblock">
  <div class="content">
    <img src="https://via.placeholder.com/150" alt="Description of the
    ↪  image"/>
  </div>
</div>
```

## Inline images

Images can be added inline with the macro `image`

Mau source

```
This is a paragraph with an image
↪  [image](https://via.placeholder.com/30,alt_text="A placeholder")
```

# Chapter 8

# Blocks

The chamber was lit by a wide shaft high in the further eastern wall; it slanted upwards and, far above, a small square patch of blue sky could be seen. The light of the shaft fell directly on a table in the middle of the room: a single oblong block, about two feet high, upon which was laid a great slab of white stone.
*J.R.R. Tolkien, "The Lord of the Rings - The Fellowship of the Ring" (1954)*

Mau has the concept of blocks, which are parts of the text delimited by fences

Mau source

```
----
This is a block
----
```

You can use any sequence of 4 identical characters to delimit a block, provided this doesn't clash with other syntax like the one used for headers or for comments

Mau source

```
++++
This is a block
++++

%%%%
This is another block
%%%%
```

If you need to insert 4 identical characters on a line for some reasons, you have to escape the first of them

Mau source

```
These are paragraphs.

\++++

Just standard paragraphs.
```

These are paragraphs.

++++

Just standard paragraphs.

---

Blocks isolate the text they contain and can apply special formatting rules to it. The idea behind blocks is that the text inside them can be formatted using special templates that apply to the whole content and not only to a single element such as a paragraph. They are therefore useful to create annotations and asides, to include source code, or to implement custom processing such as isolating headers from the main document.

In this and the next chapters we will explore blocks in depth and learn about all their features. As we will see, they can be heavily customised, but for now let's consider the default rendering of a block in HTML

Mau source

```
----
This is a block
----
```

HTML output

```
<div>
  <div class="content">
    <p>This is a block</p>
  </div>
</div>
```

In this manual, the default rendering of a block is the following

This paragraph is contained in a default block.

## Block titles

Blocks can have titles, created before the block by a line starting with a dot

Mau source

```
. The title
----
This is a block
----
```

HTML output

```
<div>
  <div class="title">The title</div>
  <div class="content">
    <p>This is a block</p>
  </div>
</div>
```

The space between the dot and the title is optional

Mau source

```
.The title
----
This is a block
----
```

## Block attributes

Blocks can have attributes, specified between square brackets before the opening fence. If attributes are specified, the first attribute is always the type of the block.

Mau source

```
[aside]
----
This is a block of type `aside`
----
```

HTML output

```
<div class="aside">
  <div class="content">
    <p>This is a block of type <code>aside</code></p>
  </div>
</div>
```

Blocks without an explicit type have the type `default` so the following two definitions give the same result

Mau source

```
1  ----
2  This is a default block
3  ----
4
5  [default]
6  ----
7  This is a default block
8  ----
```

Types are important as they can render the block in different ways. In the next chapter we will explore built-in blocks, and later we will discuss templates that allow us to further customise how blocks are rendered.

---

You can specify title and attributes in any order.

Mau source

```
. Title of the block
[aside]
----
This is a block of type `aside` and a title
----

[aside]
. Title of the block
----
This is a block of type `aside` and a title
----
```

Title and attributes are consumed by the first block that appears after them, so they don't need to be adjacent to it.

Mau source

```
. Title of the block

[aside]

----
This is a block of type `aside`
----
```

## Secondary content

Blocks have the concept of *secondary content*, which is any paragraph that is adjacent to the closing fence. This paragraph is included in the block metadata and used according to the type of block

(see the chapter about source code blocks, for example). The default block simply discards that content

Mau source

```
----
Content of the block
----
Secondary content that won't be in the output

This is not part of the block
```

Content of the block

This is not part of the block

# Chapter 9

# Blocks: built-in types and other features

Over an inner wall he saw the pinnacles of strangely shaped towerlike structures. One of these towers was built in, or projected into the court in which he found himself, and a broad stair led up to it, along the side of the wall.
*Robert E. Howard, "The Pool of the Black One" (1933)*

Mau provides some built-in block types that are rendered in a different way.

## Quotes

The simplest block type that Mau provides is called `quote`. This block's content is the quote itself, while the secondary content is the source of the quote, commonly called attribution.

Mau source

```
[quote]
----
Learn about the Force, Luke.
----
_Star Wars_, 1977
```

HTML output

```
<blockquote>
  <p>Learn about the Force, Luke.</p>
  <cite>
    <p><em>Star Wars</em>, 1977</p>
  </cite>
</blockquote>
```

You can see the output of blocks `quote` rendered at the beginning of each chapter in this book.

## Admonitions

Mau supports admonitions, special blocks that are meant to be highlighted and separated from the rest of the text. For example, admonitions can be rendered with an icon on the side, or with a different background and a title. This generally includes asides, notes, warnings, tips, and other similar blocks of text.

The block is specified as `[admonition, CLASS, ICON, LABEL]`.

Mau source

```
[admonition, note, "fa-solid fa-circle-info", "Info"]
----
This is my note
----
```

HTML output

```
<div class="admonition note">
  <i class="fa-solid fa-circle-info"/>
  <div class="content">
    <div class="title">Info</div>
    <div>
      <p>This is my note</p>
    </div>
  </div>
</div>
```

Info

This is an example of admonition rendered with a title and a specific colour for the border and the background.

## Block conditions

You can conditionally render a block according to the result of a test. The test is expressed in the form `CONDITION:VARIABLE:[VALUE]` and is passed to the block through the attribute `condition`.

Mau source

```
:render:yes

[aside, condition="if:render:yes"]
----
This will be rendered
----

[aside, condition="if:render:no"]
----
This will not be rendered
----
```

This will be rendered

You can use boolean values leaving out the `VALUE` part

```
:render:

[aside, condition="if:render:"]
----
This will be rendered
----

:!render:

[aside, condition="if:render:"]
----
This will not be rendered
----
```

This will be rendered

You can reverse the condition using `ifnot`

```
:render:

[aside, condition="ifnot:render:"]
----
This will not be rendered
----
```

Conditions can be used even without the block type, which can be used to conditionally render standard paragraphs.

```
:detailed:

[condition="if:detailed:"]
----
This will be rendered only when the variable `detailed` is true.
----

[condition="ifnot:detailed:"]
----
This will be rendered only when the variable `detailed` is false
----
```

## Block classes

You can add custom classes to a block using the attribute `classes`, which is a comma separated list of names. These classes will be then rendered according to the output format. For example, in HTML these will become CSS classes.

```
[aside, classes="myclass1,myclass2"]
----
This is a block of type `aside` with additional classes
----
```

HTML output

```
<div class="aside myclass1 myclass2 ">
  <div class="content">
    <p>This is a block of type <code>aside</code> with additional
    ↪  classes</p>
  </div>
</div>
```

## Including content

As mentioned in the chapter about basic syntax, the directive `#include` can be used at any point (at the beginning of the line), and this means that the content of the block can be generated including the content of an external file.

Mau source

```
[aside]
----
::#include:/path/to/important_aside.mau
----
```

This might be very useful for `source` blocks (see the dedicated chapter)

Mau source

```
[source, python]
----
::#include:/path/to/myscript.py
----
```

# Chapter 10

# Source code blocks

The older authorities seemed rather more helpful than the newer ones, and Armitage concluded that the code of the manuscript was one of great antiquity, no doubt handed down through a long line of mystical experimenters.

*H.P. Lovecraft, "The Dunwich Horror" (1929)*

For programmers, one of the most useful features of a markup language is source code blocks. Mau provides full support to source code highlighting, and allows you to highlight single lines and add callouts to the code to mark specific steps of the process or to add explanations.

## Basic source blocks

Verbatim paragraphs and source code can be printed using blocks of type `source`

Mau source

```
. Source code
[source]
----
This is all verbatim.

== This is not a header

[These are not attributes]
----
```

Source code

```
This is all verbatim.

== This is not a header

[These are not attributes]
```

You can specify the language of the source code, that can be used by highlighting tools.

Highlighting for TeX is not provided by default inside Mau. This book has been highlighted using minted.

Mau source

```
[source,python]
----
def header_anchor(text, level):
    """
    A simple Python function
    """

    return "h{}-{}-{}".format(
        level, quote(text.lower())[:20], str(id(text))[:8]
    )  # pragma: no cover
----
```

```python
def header_anchor(text, level):
    """
    A simple Python function
    """

    return "h{}-{}-{}".format(
        level, quote(text.lower())[:20], str(id(text))[:8]
    )  # pragma: no cover
```

## Callouts

Source blocks support callouts, that allow you to add notes to specific lines of code. Callouts are added at the end of the line between colons (e.g. `:1:`) the relative text is added to the secondary

content of the block

Mau source

```
[source,python]
----
def header_anchor(text, level)::1:
    return "h{}-{}-{}".format(
        level, quote(text.lower())[:20], str(id(text))[:8]:2:
    )  # pragma: no cover
----
1: The name of the function
2: Some memory-related wizardry
```

Callouts are not supported by the TeX package `minted`, so they are not rendered in this PDF.

The default delimiter for callouts is a colon `:`, but if that clashes with the syntax of your language you can pick a different one with the attribute `callouts`

Mau source

```
[source, python, callouts="|"]
----
def header_anchor(text, level):|1|
    return "h{}-{}-{}".format(
        level, quote(text.lower())[:20], str(id(text))[:8]|2|
    )  # pragma: no cover
----
1: The name of the function
2: Some memory-related wizardry
```

---

Callouts names are just strings, not manipulated by Mau, so you can use them out of order

Mau source

```
[source,python]
----
def header_anchor(text, level)::1:
    return "h{}-{}-{}".format(:3:
        level, quote(text.lower())[:20], str(id(text))[:8]:2:
    )  # pragma: no cover
----
1: The name of the function
2: Some memory-related wizardry
3: This is the return value
```

Callouts are not limited to digits, you can use non-numeric labels

Mau source

```
[source,python]
----
def header_anchor(text, level)::step1:
    return "h{}-{}-{}".format(:step3:
        level, quote(text.lower())[:20], str(id(text))[:8]:step2:
    )  # pragma: no cover
----
1: The name of the function
2: Some memory-related wizardry
3: This is the return value
```

Callouts do not need to have a definition

Mau source

```
[source,python]
----
def header_anchor(text, level)::1:
    return "h{}-{}-{}".format(
        level, quote(text.lower())[:20], str(id(text))[:8]:2:
    )  # pragma: no cover
----
```

And you can reference them in the text using `[class](1, "callout")`, and rendering it accordingly.

## Highlight lines

You can highlight lines using a callout with the special name `@`

Mau source

```
[source,python]
----
def header_anchor(text, level)::@:
    return "h{}-{}-{}".format(
        level, quote(text.lower())[:20], str(id(text))[:8]:@:
    )  # pragma: no cover
----
```

# Chapter 11

# Footnotes and references

> And then I told him my story as I have written it here, omitting only any reference to my love for Dejah Thoris. He was much excited by the news of Helium's princess and seemed quite positive that she and Sola could easily have reached a point of safety from where they left me.
>
> *Edgar Rice Burroughs, "A Princess of Mars" (1912)*

Footnotes are extremely useful to provide optional details to parts of the text, or to mention source material like books or papers. Mau has full support for footnotes with a syntax that comes from Markua (Leanpub's Markdown dialect). Mau also expands the concept into references, that are a generalisation of footnotes.

## Footnotes

Footnotes in Mau work through a macro, a block type, and a command.

The block type `footnote` contains the text of the footnote. The name of the footnote is a mandatory argument

Mau source

```
[footnote, mynote]
----
This is the text of my footnote.
----
```

Inside the block you can use Mau syntax, such as styles or links

Mau source

```
[footnote, mynote]
----
_This_ is the *text* of my footnote.
----
```

The macro `footnote` appears in a paragraph and represents the reference to the footnote, usually being rendered as a superscript small number. The only argument of the macro is the name of the referenced footnote.

Mau source

```
There are several markup languages available nowadays[footnote](mynote).
```

The command `::footnotes:` inserts the list of footnotes and is useful for those output formats like HTML that do not include footnotes automatically.

Mau source

```
1  There are several markup languages available
   ↪  nowadays[footnote](mynote).
2
3  [footnote, mynote]
4  ----
5  For example:
6
7  * [link]("HTML", https://en.wikipedia.org/wiki/HTML)
8  * [link]("Markdown", https://en.wikipedia.org/wiki/Markdown)
9  * [link]("TeX", https://en.wikipedia.org/wiki/TeX)
10 ----
11
12 ::footnotes:
```

There are several markup languages available nowadays[a].

[a] For example:

- https://en.wikipedia.org/wiki/HTML
- https://en.wikipedia.org/wiki/Markdown
- https://en.wikipedia.org/wiki/TeX

Please note that the way footnotes are managed can be different according to the output format. If you are producing HTML output you need to insert the command as HTML has no concept of footnotes. TeX, instead, automatically creates the list of footnotes at the bottom of each page, so in that case you don't need to render the list explicitly (see the chapter about templates).

## References

References are a generalisation of footnotes, and in principle they work in the same way through the macro `reference`, the block of type `reference`, and the command `references`.

Unlike footnotes that have just a name, though, references have a type, a name, and a category. The command `references` can be used to print them out selectively.

So, for example, a reference of type `links` with name `markup` is defined as

Mau source

```
There are many markup languages[reference](links, markup).

[reference, links, markup]
----
For example:

* [link]("HTML", https://en.wikipedia.org/wiki/HTML)
* [link]("Markdown", https://en.wikipedia.org/wiki/Markdown)
* [link]("TeX", https://en.wikipedia.org/wiki/TeX)
----
```

If you include a category you need to specify it only in the macro

Mau source

```
There are many markup languages[reference](links, markup, wikipedia).

[reference, links, markup]
----
For example:

* [link]("HTML", https://en.wikipedia.org/wiki/HTML)
* [link]("Markdown", https://en.wikipedia.org/wiki/Markdown)
* [link]("TeX", https://en.wikipedia.org/wiki/TeX)
----
```

The command `references` requires the type, and optionally accepts the category and the name

Mau source

```
::references:links

::references:links,wikipedia

::references:links,wikipedia,markup
```

Each reference is uniquely identified by the tuple `(type, name)`, and the category can act as a subtype.

Footnotes can be considered references with the type `footnote` and no category. The code

Mau source

```
There are several markup languages available
↪  nowadays[footnote](mynote).

[footnote, mynote]
----
For example:

* [link]("HTML", https://en.wikipedia.org/wiki/HTML)
* [link]("Markdown", https://en.wikipedia.org/wiki/Markdown)
* [link]("TeX", https://en.wikipedia.org/wiki/TeX)
----

::footnotes:
```

is equivalent to

Mau source

```
There are several markup languages available
↪  nowadays[reference](footnote, mynote).

[reference, footnote, mynote]
----
For example:

* [link]("HTML", https://en.wikipedia.org/wiki/HTML)
* [link]("Markdown", https://en.wikipedia.org/wiki/Markdown)
* [link]("TeX", https://en.wikipedia.org/wiki/TeX)
----

::references:footnote
```

Keep in mind, however, that footnotes and references are rendered with different templates. In particular, output formats like TeX have a native support for footnotes but don't know anything about references.

# Chapter 12

# Block engines and custom definitions

- “Impact minus twenty seconds, guys. . . ” said the computer.

- “Then turn the bloody engines back on!” bawled Zaphod.

- “OK, sure thing, guys,” said the computer.
*Douglas Adams, "The Hitchhiker’s Guide to the Galaxy" (1979)*

So far we saw how Mau treats standard blocks, the built-in types `quote`, `admonition`, and `source`, and how to render blocks conditionally. Blocks have another advanced feature called `engine`, which rule the way Mau processes the content of the block.

## Block engines

In blocks, an `engine` is the way Mau processes content and attributes to create the values eventually passed to the template used to render the block.

When a block doesn’t define a specific engine the `default` one is used. Mau currently defines six engines:

- `default`
- `source`
- `raw`
- `mau`
- `footnote`
- `reference`

## Default

The engine `default` processes the content of the block as Mau code using the variables defined previously in the document. It adds the headers found in the block to the global TOC and the footnotes to the list of the document footnotes.

This engine is used when no other engine is defined explicitly and is useful every time we need to customise the way the content is rendered in the final format but we want to keep the content as part of the document.

## Raw

The engine `raw` is useful every time we want to include text in the output format directly, as this engine doesn't process the content at all. For example, if we are converting the Mau source in HTML and we want to add custom HTML code we can do it with

Mau source

```
[aside, engine=raw]
----
<div>This is a custom div written directly in HTML</div>
----
```

Please note that while the content of the block is not processed by Mau it is still rendered as any other block through templates. See the following chapters to find out how to include custom content without any wrapper.

## Mau

As we saw previously, the engine `default` processes the content as Mau code and adds variables, headers and footnotes in the current document. The engine `mau` does the same, but treats it as isolated content, without loading the variables defined in the main document, and without adding headers and footnotes to the relative lists.

Using a command `::toc:` in a block rendered by the engine `mau`, for example, will include only the headers defined in the block itself.

Mau source

```
[aside,engine=mau]
----
= Main section

== Secondary section

== Another secondary section

=== A very specific section

::toc:
----
```

Please note that even with this engine the content of the block is still rendered in the current document. This means that with output formats like TeX that create their own TOC automatically such headers will still be part of the main document.

## Source

The engine that processes source blocks is aptly called `source`. This engine scans the primary content for callouts and looks for their optional definitions in the secondary content.

Blocks of type `source` are a shortcut notation to use the engine `source`. The notation

Mau source

```
[source]
----
Some source code
----
```

is equivalent to

Mau source

```
[default, engine=source]
----
Some source code
----
```

The explicit use of the engine `source` is useful to create custom source blocks that can behave like the built-in one but can be rendered in different ways. The way to do it will be shown in the chapter about templates.

### Footnotes and references

The two engines `footnotes` and `references` process the content independently from the main document, keeping headers and variables isolated, and store the content into a global list that can be rendered with the relative commands `::footnotes:` and `::references:`. As happened for `source`, blocks of type `footnote` and `reference` are actually default blocks using the engine with the same name behind the scenes.

## Custom block definitions

As we saw in previous chapters and in the previous sections, blocks have many attributes that you can set on them. In the future chapters about templates we will also see how you can create and use custom attributes.

Setting the same parameters over and over can become tedious and error prone, so Mau provides a way to define blocks through the command `defblock`.

Mau source

```
::defblock:python, default, engine=source, language=python

[python]
----
a = 5
----
```

which equivalent to

Mau source

```
[default, engine=source, language=python]
----
a = 5
----
```

and is rendered as

```
a = 5
```

The syntax of a block definition is

Mau source

```
::defblock:ALIAS, BLOCKTYPE, ATTRIBUTE1, ATTRIBUTE2, ...
```

where `ALIAS` is the name you will use in your Mau document with `[ALIAS]` and `BLOCKTYPE` is the actual block type that corresponds to that alias.

The attributes can have a default value or not. If they have no value they need to be specified when using the alias. For example the following definition

```
::defblock:mysource, aside, language, engine=source
```

defines `mysource` as an alias for the block type `aside`, and requires the attribute `language` to be provided. The attribute `engine`, instead, has a default value and can be omitted. So, the following two blocks are both valid

```
[mysource, python]
----
a = 5
----

[mysource, python, engine=raw]
----
a = 5
----
```

while omitting a value for `language` will result in a syntax error

```
// This block is not valid
[mysource]
----
a = 5
----
```

Block definitions are static aliases, so it is perfectly fine to write a definition like

Mau source

```
::defblock:myblock, myblock, engine=raw, classes="myclass1,myclass2"
```

which allows to use the type `myblock` in conjunction with a default set of arguments.

## Predefined blocks

As we saw in the previous sections, Mau defines provides some aliases out of the box. While these are created directly in the Python soruce code, they are equivalent to the following definitions

Mau source

```
::defblock:source, default, language=text, engine=source
::defblock:footnote, default, name, engine=footnote
::defblock:reference, default, type, name, engine=reference
::defblock:admonition, admonition, class, icon, label
```

The built-in block type `quote` is not an alias and doesn't contain additional attributes, so it works just like a `default` block.

# Chapter 13

# Basic templates

'That's the idea,' replied Norton. 'This may be an indexed catalogue for 3-D images—templates—solid blueprints, if you like to call them that.'
*Arthur C. Clarke, "Rendezvous with Rama" (1973)*

So far, Mau provides features similar to Markdown, Asciidoc, and other markup languages. The real power of Mau, however relies in its use of templates to render the output of the syntax.

Everything you saw in the previous chapters is rendered as described thanks to the default Jinja templates provided by the visitors, but such templates can be easily customised to provide the output you prefer.

## A simple example

To start off on our journey with templates let's have a look a the simplest ones. You know that the Mau provides styles like

Mau source

```
Stars identify *strong* text.
```

This piece of text is processed by Mau and the following Abstract Syntax Tree (AST) is created

Abstract Syntax Tree

```yaml
data:
  args: []
  content:
  - data:
      args: []
      content:
        data:
          content:
          - data:
              type: text
              value: 'Stars identify '
          - data:
              content:
                data:
                  content:
                  - data:
                      type: text
                      value: strong
                  type: sentence
              type: style
              value: star
          - data:
              type: text
              value: ' text.'
          type: sentence
      kwargs: {}
      tags: []
      type: paragraph
  kwargs: {}
  tags: []
  type: document
```

You can always visualise the AST using the output format `dump` on the command line

```
mau -f dump -i input.mau -o -
```

The AST is then processed by the chosen visitor, that will render each node. In this case the nodes are of type `text`, `sentence`, `style`, `paragraph`, and `document`.

The `style` node in the previous AST is

```
- data:
  content:
    [...]
  type: style
  value: star
```

and the default HTML template for it is

```
star.html: '<strong>{{ content }}</strong>'
```

The output of the whole text will be

HTML output

```
<p>Stars identify <strong>strong</strong> text.</p>
```

where you can see the template `star.html` in action.

If we changed the template to

```
star.html: '<span class="italic">{{ content }}</span>'
```

the result would be

HTML output

```
<p>Stars identify <span class="italic">strong</span> text.</p>
```

As you can see the names of the templates include an extension even though they are defined in a Python dictionary. This is done as templates can be provided as stand-alone files as well, and keeping a consistent naming simplifies their management.

## Template names

Each element of a Mau text is rendered using a specific template. The template can be unique for the element (e.g. `underscore.html` for the style `underscore`) or be selected among several options.

For example, blocks templates take into account the engine and the type. The block

Mau source

```
[aside]
----
This is a block of type `aside`
----
```

will be rendered in HTML with the first of the following templates that is available:

- `block-default-aside.html`
- `block-default.html`
- `block-aside.html`
- `block.html`

You will find a detailed list of templates in the description of each node in the relative chapter. The list of templates for each node is decided by the visitor, so different visitors can provide different options.

# How to define templates

Mau templates can be provided as files (one per template) or though a YAML configuration file.

## Configuration file

Mau templates can be defined in the YAML configuration file using the key `custom_templates`. For example

```
---
target_format: html
custom_templates:
  underscore.html: '<span class="bold">{{ content }}</span>'
  star.html: '<span class="italic">{{ content }}</span>'
```

This is a good solution for simple templates, but it might quickly become unmanageable if your templates include a lot of Jinja operators.

### Individual files

You can store templates in individual files inside a directory and pass the latter to Mau through the configuration file with the key `templates_directory`.

```
---
target_format: html
templates_directory: templates
```

Each template is stored in a file with the appropriate name and contains only the code of the template, e.g.

templates/underscore.html

```
<span class="bold">{{ content }}</span>
```

This is a good solution for complex templates, as it doesn't require to quote the text. You can also benefit from syntax highlighting in your editor while writing the template.

## Templates syntax

Mau provides a base visitor class `BaseVisitor` that produces the YAML AST I showed previously. Mau also provides a `JinjaVisitor` class that uses the powerful template engine Jinja.

The current visitor plugins are created as subclasses of `JinjaVisitor` but nothing stops you from creating a custom visitor that uses another engine.

The variables available to each Jinja template can be found in the description of each node in the next chapter.

# Chapter 14

# Elements and nodes

Zane noted how artistically the elements of both carpet and machine had been integrated to make the device unidentifiable; this was a symbolic example, not a literal one. It had also been hastily done, for Luna had been home only a few hours.
*Piers Anthony, On a Pale Horse (1983)*

This chapter contains the full documentation of all Mau elements. For each element you will find:

- A recap of the syntax.
  - The scope of the node can be `inline` or `page`. `inline` nodes appear in paragraphs, mixed with normal text, while `page` nodes have to appear on separate lines of the document.
  - The type of the node is a string that uniquely identifies it in Mau.
- A short description.
- The names of the templates that the `JinjaVisitor` class uses to render it in order of priority (without the format extension).
- The documentation about the fields passed to the Jinja template with the type and a short description
  - `BOOLEAN`: a Python boolean that can be used directly in Jinja conditional statements.
  - `INTEGER`: a Python integer.
  - `LIST[TYPE]`: a list of values of the given `TYPE`.
  - `STRING`: a Python string that contains pure text.

  - `TARGET`: the target format. For example when rendering in HTML a field of type `TARGET` will insert some HTML.
  - `VALUE1 | VALUE2 | VALUE3`: the value can only be one of the listed ones.
  - `SPECIAL`: usually a more complex Python datatype like a dictionary or a list of tuples. See the description of the node for more details.
  - The fields `args`, `kwargs`, and `tags` have always the same type: a list of strings for `args` and `tags`, a dictionary for `kwargs`.
  - A template receives always the field `type` that contains the type of the node.

- The default HTML and LaTeX templates. Sometimes a format doesn't specify a certain template, and this situation will be marked as `EMPTY`. Sometimes templates contain explicit newlines `\n`, which has been left to clarify that a new line is created of that the templates terminates with an empty line.

- Some examples.

To simplify the use of this chapter elements are listed in alphabetical order instead of being grouped per topic like in the rest of the book.

# Block

This element represents a generic block.

### Syntax

Type: `page`

Mau source

```
[BLOCKTYPE]
----
CONTENT
----
```

### Templates

- `block-{engine}-{blocktype}`
- `block-{engine}`
- `block-{blocktype}`
- `block`

### Fields

- `blocktype (STRING)`: the type of this block
- `content (TARGET)`: the primary content (Mau text between fences)
- `secondary_content (TARGET)`: the secondary content (Mau text just below the closing fence
- `classes (LIST[STRING])`: a list of classes assigned to this block
- `title (STRING)`: a title specified before this block
- `engine (STRING)`: the engine that Mau used to process the block
- `preprocessor (STRING)`: the preprocessor (currently not in use)
- `args`

- kwargs
- tags

### Default templates

HTML

```
<div class="{{ blocktype }}{% if classes %} {{ classes }}{% endif %}">
  {% if title %}<div class="title">{{ title }}</div>{% endif %}
  <div class="content">{{ content }}</div>
</div>
```

LaTeX

```
{{ content }}\n
```

Markua

```
{{ content }}
```

### Examples

Mau source

```
[aside, classes="custom,centred"]
----
This is the primary content of a block of type `aside` and with two
↪  classes `custom` and `centred`.
----
This is the secondary content
```

# Callout

This element renders a callout attached to each line of code.

See: Source (block)

### Templates

- `callout`

### Fields

- `marker (STRING)`: the label of this callout

### Default templates

HTML

```
<span class="callout">{{ marker }}</span>
```

LaTeX

```
EMPTY
```

Markua

```
EMPTY
```

# Callout entry

This element renders an element in the list of callouts. This list is usually presented at the end of a source block to show the text associated with each callout.

See: Source (block), Callout

### Templates

- `callouts_entry`

### Fields

- `marker (STRING)`: the label of this callout
- `value (STRING)`: the text associated with this callout

### Default templates

HTML

```
<tr>
<td><span class="callout">{{ marker }}</span></td>
<td>{{ value }}</td>
</tr>
```

LaTeX

```
EMPTY
```

Markua

```
EMPTY
```

# Caret (style)

This style renders the text between two carets according to the template. The default template renders the text in superscript.

### Syntax

Scope: `inline` Type: `style`

Mau source

```
^TEXT^
```

### Templates

- `caret`

### Fields

- `value (STRING)`: the style type (`caret`)
- `content (TARGET)`: the content of the node

### Default templates

HTML

```
<sup>{{ content }}</sup>
```

LaTeX

```
\textsuperscript{ {{-content-}} }
```

Markua

```
^{{ content }}^
```

### Examples

Mau source

```
This text is super^script^.
```

Rendered

This text is super$^{\text{script}}$.

# Class (macro)

This macro attaches one of more classes to a piece of text. At the moment, the default LaTeX output doesn't use classes.

### Syntax

Scope: `inline` Type: `class`

Mau source

```
[class](text, "class1,class2,...")
```

### Templates

- `class`

### Fields

- `classes [LIST(STRING)]`: the list of classes
- `content (TARGET)`: the content of the node

### Default templates

HTML

```
<span class="{{ classes | join(' ') }}">{{ content }}</span>
```

LaTeX

```
{{content}}
```

Markua

```
{{ content }}
```

### Examples

Mau source

```
This text is [class]("coloured", "green") thanks to a class.
```

# Content

This element is a very generic content loader, which has a specific implementation only for the type `image` (see "Image (content)"). It accepts arguments and consumes a title, so it can be used to achieve custom effects without involving blocks, whose syntax is more invasive.

### Syntax

Scope: `page` Type: `content`

Mau source

```
<< TYPE:ARGS,KWARGS,TAGS
```

### Templates

- `content-{content_type}`
- `content`

### Fields

- `content_type (STRING)`: the type of this content
- `title (TARGET)`: an optional title for the content
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
EMPTY
```

LaTeX

```
EMPTY
```

Markua

```
EMPTY
```

### Examples

Mau source

```
```

# Document

This element is created automatically by Mau and wraps the whole document.

### Syntax

Type: `page`

### Templates

- `document`

### Fields

- `content (TARGET)`: the content of the document

### Default templates

HTML

```
<html>
  <head></head>
  <body>{{ content }}</body>
</html>
```

LaTeX

```
{{ content }}
```

Markua

```
{{ content }}
```

# Footnote (block)

This block defines the content of a footnote and has to be paired with a macro `footnote` with the same name. Footnotes have a definition (the text of the footnote) and a reference (usually the number of the footnote in a paragraph). When footnotes are inserted in the document, the content of each footnote in the list is represented by this element. Since LaTeX manages footnotes automatically, the default template in that case is empty.

### Syntax

Scope: `page` Type: `footnotes_entry`

Mau source

```
[footnote, NAME]
----
CONTENT
----
```

### Templates

- `footnotes_entry`

### Fields

- `content (TARGET)`
- `number (STRING)`: The number of this footnote
- `reference_anchor (STRING)`: The anchor to the footnote reference
- `definition_anchor (STRING)`: The anchor to the footnote definition

### Default templates

HTML

```
<div id="{{ definition_anchor }}">
  <a href="#{{ reference_anchor }}">{{ number }}</a> {{ content }}
</div>
```

LaTeX

```
EMPTY
```

Markua

```
[^footnote_{{ reference_anchor }}]: {{ content }}
```

### Examples

Mau source

```
[footnote, mynote]
----
This is the body of the note called `mynote`. It will be referenced
↪  somewhere in the text through the macro `footnote`.
----
```

# Footnote (macro)

This element defines the reference to a footnote. It has to be paired with a block `footnote` that defines the content of the footnote.

Footnote macros and blocks are collected and processed as a whole, which is why this node can provide the content.

The anchors to the reference and the definition are unique strings in the document that can be used to set up internal links for formats like HTML that do not provide this service out of the box like LaTeX does.

### Syntax

Scope: `inline` Type: `footnote`

Mau source

```
[footnote](NAME)
```

### Templates

- `footnote`

### Fields

- `content (TARGET)`: The content of the footnote defined by the companion block
- `number (STRING)`: The number of this footnote
- `reference_anchor (STRING)`: The anchor to this reference
- `definition_anchor (STRING)`: The anchor to the definition

### Default templates

HTML

```
<sup>
[<a id="{{ reference_anchor }}" href="#{{ definition_anchor }}">{{ number
↪  }}</a>]
</sup>
```

LaTeX

```
\footnote{ {{-content-}} }
```

Markua

```
[^footnote_{{ reference_anchor }}]
```

### Examples

Mau source

```
This paragraph contains a footnote[footnote](mynote). It also contains
↪  other sentences.
```

# Footnotes (command)

This command inserts the list of footnote definitions. Since LaTeX automatically puts footnotes at the bottom of each page, the default template in that case is empty.

### Syntax

Scope: `page` Type: `command_footnotes`

Mau source

```
::footnotes:ARGS, KWARGS, TAGS
```

### Templates

- `footnotes`

### Fields

- `entries (LIST[TARGET])`: this is the list of footnotes, each being the rendering of a footnote entry.
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<div id="_footnotes">{{ entries }}</div>
```

LaTeX

```
EMPTY
```

Markua

```
{{ entries }}
```

### Examples

Mau source

```
::footnotes:
```

# Header

This element renders a section header and stores it in the ToC.

### Syntax

Scope: `page` Type: `header`

Mau source

```
= Header level 1
== Header level 2
=== Header level 3
[...]
```

### Templates

- `header{level}`
- `header`

### Fields

`value (STRING)`: the text of the header `level (INTEGER)`: the level of the header, 1 being the highest. `anchor (STRING)`: the anchor of the header if the template wants to set up cross-references between it and the ToC `args` `kwargs` `tags`

### Default templates

HTML

```
<h{{ level }} id="{{ anchor }}">{{ value }}</h{{ level }}>
```

LaTeX

```
\{{command}}{ {{-value-}} }
\n
```

Markua

```
{{ '#' * level }} {{ value }}
```

### Examples

Mau source

```
= An important section

== A less important one
```

# Horizontal rule

This element inserts a horizontal rule. The rule can be optionally given arguments for further customisation.

### Syntax

Scope: `page` Type: `horizontal_rule`

Mau source

```
---
```

Mau source

```
---:ARGS,KWARGS,TAGS
```

### Templates

- `horizontal_rule`

### Fields

- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<hr>
```

LaTeX

```
\rule{\textwidth}{0.5pt}
\n
```

Markua

```
* * *
```

### Examples

Mau source

```
---
```

Rendered source

---

Mau source

```
[break]
---
```

Mau source (template)

```
{% if 'break' in args %}
\pagebreak
{% else %}
\vspace{24pt}\noindent\hfil\rule{0.5\textwidth}{.4pt}\vspace{24pt}\hfil
{% endif %}
```

# Image (content)

This element inserts an image int he document. The image is inserted as a page element, for images included in paragraphs see "Image (macro)".

### Syntax

Type: `page`

Mau source

```
<< image:URI, ALT_TEXT, CLASSES, ARGS, KWARGS, TAGS
```

### Templates

- `content_image`

### Fields

- `uri (STRING)`: the URI of the image file
- `alt_text (STRING)`: alternative text to show if the image cannot be loaded
- `classes (LIST[STRING])`: list of classes to append to the target element
- `title (STRING)`: title (caption) of the image
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<div class="imageblock">
<div class="content">
<img src="{{ uri }}"{% if alt_text %} alt="{{ alt_text }}"{% endif %} />
{% if title %}<div class="title">{{ title }}</div>{% endif %}
</div></div>
```

LaTeX

```
\begin{figure}[h]
{% if title %}\caption{ {{-title-}} }{% endif %}
\centering
\includegraphics[width=\textwidth]{ {{-uri-}} }
\end{figure}
\n
```

Markua

```
{% if alt_text %}{alt: "{{ alt_text }}"}\n{% endif %}![{{ title }}]({{
↪  uri }})
```

### Examples

Mau source

```
<< image:../images/cover.jpg, "The cover of the book", "centered",
```

# Image (macro)

Through this element, Mau inserts an image in a paragraph.

### Syntax

Scope: `inline` Type: `image`

Mau source

```
[image](URI, ALT_TEXT, WIDTH, HEIGHT)
```

### Templates

- `inline_image`

### Fields

- `uri (STRING)`: the URI of the image in a format understood by the target processor.
- `alt_text (STRING)`: the alternative text used if the image cannot be loaded.
- `width (STRING)`: The width of the image
- `height (STRING)`: The height of the image

### Default templates

HTML

```
<span class="image">
<img src="{{ uri }}"{%if alt_text %} alt="{{ alt_text }}"{% endif %}>
</span>
```

LaTeX

```
\includegraphics{ {{-uri-}} }
```

Markua

```
![{{ alt_text }}]({{ uri }})
```

### Examples

Mau source

```
This paragraph contains a little emoji [image](TODO).
```

# List

This element renders a list of items. The list can be ordered or unordered, and items can be nested. Each item in the template is either a single item (rendered with the relative template) or a sublist (rendered with the same template as the whole list).

### Syntax

Scope: `page` Type: `list`

Mau source

```
* Unordered list item
** Unordered list item of level 2
*** Unordered list item of level 3
   *** Spaces are ignored before list item symbols
   *** So you are free to use them or not
[...]
# Ordered list item
## Ordered list item of level 2
### Ordered list item of level 3
[...]
```

### Templates

- `list`

### Fields

- `ordered (BOOLEAN)`: whether the list is ordered or not
- `items (CONTENT)`: a list containing either items or sublists
- `main_node (BOOLEAN)`: whether this is the main list of a sublist
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<{% if ordered %}ol{% else %}ul{% endif %}{% if kwargs.start %}start={{
↪  kwargs.start }}{% endif %}>
{{ items }}
</{% if ordered %}ol{% else %}ul{% endif %}>
```

LaTeX

```
{% if not main_node %}\n{% endif %}
{% if ordered %}\begin{enumerate}{% else %}\begin{itemize}{% endif %}

{{ items }}
{% if ordered %}\end{enumerate}{% else %}\end{itemize}{% endif %}
{% if main_node %}\n\n{% endif %}"
```

Markua

```
{{ items }}{% if main_node %}\n{% endif %}
```

## Examples

Mau source

```
* This is an unordered list
* That contains an ordered list
## The ordered list
## Is a sublist, though
* Lists can have
** Many levels
*** As you would expect
```

Rendered source

- This is an unordered list
- That contains an ordered list
    1. The ordered list
    2. Is a sublist, though
- Lists can have
    - Many levels
        * As you would expect

# List item

This element renders a single item in a list (see `List`).

### Templates

- `list_item`

### Fields

- `level (INTEGER)`: The level of nesting (1 is the first level).
- `content (target)`: the content of this item

### Default templates

HTML

```
<li>{{ content }}</li>
```

LaTeX

```
\item {{ content }}
\n"
```

Markua

```
{% if not main_node %}
{% endif %}{{ ' ' * level }}* {{ content }}
```

# Link (macro)

This element represents a hypermedia link to a URL.

### Syntax

Scope: `inline` Type: `link`

Mau source

```
[link](TARGET, TEXT)
```

### Templates

- `link`

### Fields

- `target (STRING)`: The target URL
- `text (STRING)`: The text of the link

### Default templates

HTML

```
<a href="{{ target }}">{{ text }}</a>
```

LaTeX

```
\href{ {{-target-}} }{ {{-text-}} }
```

Markua

```
{% if text!=target %}[{{ text }}]({{ target }}){% else %}<{{ target }}>{%
↪  endif %}
```

### Examples

Mau source

```
A lot of articles can be found on [link]("en.wikipedia.org/",
↪  "Wikipedia").
```

Rendered source

A lot of articles can be found on Wikipedia.

# Paragraph

This element represents a paragraph of text. Individual paragraphs must be separated by an empty line. Paragraphs can be given arguments to further customise them.

### Syntax

Scope: `page` Type: `paragraph`

Mau source

```
Any line of text that doesn't match another page element.
```

Mau source

```
[ARGS, KWARGS, TAGS]
This is a paragraph with arguments.
```

### Templates

- `paragraph`

### Fields

- `content (TARGET)`: the content of the paragraph
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<p>{{ content }}</p>
```

LaTeX

```
{{ content }}
\n\n
```

Markua

```
{{ content }}\n
```

### Examples

Mau source

```
This is a paragraph.

This is another paragraph.
```

Rendered source

This is a paragraph.

This is another paragraph.

# Source (block)

This element is used to include source code. The primary content of the block is interpreted verbatim aside from "callouts" that are labels that can be attached to lines of the text and that can be referenced later in the document.

The block supports highlighting of the source code through Pygments and the standard components of blocks like titles, unnamed and named arguments, and tags.

There are three elements rendered by templates: the block itself, the callout label attached to a line, and each callout entry (similar to what happens with foornotes).

Mau source

```
def add(a, b): 1 <-- This is the callout label
    return a+b
    ^
    |
    This is the primary content

1: A function to add numbers <-- This is the callout entry
```

Callouts are saved in two lists, `markers` and `callouts`. `markers` is a list of tuples `(linenum, text)`, where `linenum` is the number of a line, and `text` is the label assigned to that callout. `callouts` is a list of callout entries, rendered through the specific template.

Code is provided as a list of tuples `(line, callout)`. The variable `line` contains the line of code in the target format (possibly highlighted) and `callout` is either the rendered form of a callout (using the relative template) or `None`.

See: Callout entry, Callout

### Syntax

Type: `page`

Mau source

```
[source, CALLOUTS, ARGS, KWARGS, TAGS]
----
CONTENT
----
```

### Templates

- `source-{node.blocktype}-{node.language}`
- `source-{node.language}`
- `source-{node.blocktype}`
- `source`

### Fields

- `language (STRING)`: the language of the code contained in this block
- `callouts (LIST[TARGET])`: a list of callout entries
- `markers (SPECIAL)`: list of markers (see description)
- `highlights (LIST[INTEGER])`: list of lines that have to be highlighted
- `delimiter (STRING)`: callouts delimiter
- `code (SPECIAL)`: code and callouts (see description)
- `title (STRING)`: title of this block
- `classes (LIST[STRING])`: a list of classes assigned to this block
- `preprocessor (STRING)`: the preprocessor (currently not in use)
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<div class="{{ blocktype }}">
   {% if title %}<div class="title">{{ title }}</div>{% endif %}
   <div class="content">
     {% for line, callout in code %}{{ line }}{% if callout %} {{
     ↪  callout }}{% endif %}\n{% endfor %}
   </div>
   {% if callouts %}
   <div class="callouts">
     <table>
       <tbody>
         {% for callout_entry in callouts %}{{ callout_entry }}{%
         ↪  endfor %}
       </tbody>
     </table>
   </div>
   {% endif %}
</div>
```

LaTeX

```
{{ code }}\n
```

Markua

```
{% if title %}{caption: "{{ title }}"}\n{% endif %}
``` {% if language %}{{ language }}{% endif %}
{% for line, _ in code %}{{ line }}\n{% endfor %}
```\n
```

## Examples

Mau source

```
[source, python]
----
def add(a, b):
    return a+b
----
```

Rendered source

```
def add(a, b):
    return a+b
```

# Star (style)

This style renders the text between two stars according to the template. The default template renders the text as bold.

### Syntax

Scope: `inline` Type: `style`

Mau source

```
*TEXT*
```

### Templates

- `star`

### Fields

- `value (STRING)`: the style type (`star`)
- `content (TARGET)`: the content of the node

### Default templates

HTML

```
<strong>{{ content }}</strong>
```

LaTeX

```
\textbf{ {{-content-}} }
```

Markua

```
**{{ content }}**
```

### Examples

Mau source

```
This text is *bold*.
```

Rendered

This text is **bold**.

# Tilde (style)

This style renders the text between two tildes according to the template. The default template renders the text as subscript.

### Syntax

Scope: `inline` Type: `style`

Mau source

```
~TEXT~
```

### Templates

- `tilde`

### Fields

- `value (STRING)`: the style type (`tilde`)
- `content (TARGET)`: the content of the node

### Default templates

HTML

```
<sub>{{ content }}</sub>
```

LaTeX

```
\textsubscript{ {{-content-}} }
```

Markua

```
~{{ content }}~
```

### Examples

Mau source

```
This text is sub~script~.
```

Rendered

This text is sub$_{\text{script}}$.

# TOC (command)

This command renders the list of all headers (Table of Contents). The argument `exclude_tag` can be used to remove headers with that tag.

### Syntax

Scope: `page` Type: `command_toc`

Mau source

```
::toc:ARGS, KWARGS, TAGS
```

### Templates

- `toc`

### Fields

- `entries (LIST[TARGET])`: this is the list of headers, each being the rendering of a TOC entry.
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<div>{% if entries%}<ul>{{ entries }}</ul>{% endif %}</div>
```

LaTeX

```
EMPTY
```

Markua

```
EMPTY
```

### Examples

Mau source

```
# Header in ToC

## Subheader in ToC

::toc:
```

Mau source

```
# Header in ToC

[#notoc]
## Subheader not in ToC

::toc:exclude_tag=notoc
```

# ToC entry

This element renders an element in the TOC.

See: TOC (command)

### Templates

- `toc_entry`

### Fields

- `value (STRING)`: the header contained in this element
- `anchor (STRING)`: the anchor of the header
- `children (TARGET)`: the rendered list of children (rendered as the whole TOC)
- `args`
- `kwargs`
- `tags`

### Default templates

HTML

```
<li>
  <a href="#{{ anchor }}">{{ value }}</a>
  {% if children %}<ul>{{ children }}</ul>{% endif %}
</li>
```

LaTeX

```
EMPTY
```

Markua

```
EMPTY
```

# Underscore (style)

This style renders the text between two underscores according to the template. The default template renders the text as italic.

### Syntax

Scope: `inline` Type: `style`

Mau source

```
_TEXT_
```

### Templates

- `underscore`

### Fields

- `value (STRING)`: the style type (`underscore`)
- `content (TARGET)`: the content of the node

### Default templates

HTML

```
<em>{{ content }}</em>
```

LaTeX

```
\textit{ {{-content-}} }
```

Markua

```
*{{ content }}*
```

### Examples

Mau source

```
This text is _italic_.
```

Rendered

This text is *italic*.

# Verbatim (style)

This style treats the text inside as verbatim, and is useful to render code inside paragraphs.

### Syntax

Scope: `inline` Type: `verbatim`

Mau source

```
`TEXT`
```

### Templates

- `verbatim`

### Fields

- `value (STRING)`: the verbatim text

### Default templates

HTML

```
<code>{{ value }}</code>
```

LaTeX

```
\texttt{ {{-value-}} }
```

Markua

```
`{{ value }}`
```

### Examples

Mau source

```
This paragraphs contains `verbatim text`.
```

Rendered

This paragraphs contains `verbatim text`.

## Macro

TODO