Leonardo Giordani

The Mau Book

4th Edition - 2024

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.

As you can see, the system is production-ready, and you can start using it today.

Features

Mau's main features are

- 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)

Mau versions

This documentation covers Mau version 4.x.

Previous versions are not supported any more and won't receive any bug fixes.

How to contribute

Mau is completely open-source, so you can contribute to the Mau ecosystem opening issues and creating PRs. The main repositories can be found at `https://github.com/Project-Mau`. Contributions are more than welcome!

In particular, you might be interested in:

- The core system: `https://github.com/Project-Mau/mau`
- The HTML visitor: `https://github.com/Project-Mau/mau-html-visitor`
- This documentation: `https://github.com/Project-Mau/maubook`

If you find Mau useful, please spread the word and share the link to this documentation.

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)

This chapter details 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. However, these are 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
```

The option `-f` is used to specify the output format, and the only output available out of the box is `yaml`, which prints out the Abstract Syntax Tree using YAML. To render Mau source code you need a *visitor plugin*.

At the moment there are two plugins available: the HTML visitor and the TeX visitor. 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-aae2">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. A complete description of the configuration file can be found in Configuration. You can learn more about variables in Mau documents in Variables.

Using Mau programmatically

You can use Mau programmatically in your Python code. See Python Interface to know more about the API.

That chapter also details how to use Mau to write posts and pages in Pelican through a specific plugin.

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 other 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 this documentation will describe everything assuming the reader doesn't know any markup language.

The documentation will refer to generic input and output, meaning respectively the Mau source and the final result.

Typographic conventions

The source code in a block will be shown like this

Mau source

```
This is Mau source code
```

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

Remember that both Markdown and Asciidoctor have a fixed rendering, while the output in Mau is ruled by templates. In the first chapters what you will see in the rendered boxes is the result of Mau's default templates, but those can be overridden at any time, as detailed by later chapters.

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. 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.

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 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.

while multi-line comments are surrounded by four slashes on a separate line

Mau source

```
This is a sentence.

And another sentence.

////
This is also a comment
but it's spread on
multiple lines
for fun and to be a
little more readable
////
```

This is a sentence.

And another sentence.

Include other files

You can include 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 include files anywhere in the document. The directive, however, cannot be surrounded by other text. Therefore, you cannot include the content of a file in the middle of a paragraph or in a header.

Chapter 2

Mau Components

> The electric mechanism, within its compellingly authentic-style gray pelt, gurgled and blew bubbles, its vidlenses glassy, its metal jaws locked together. This had always amazed him, these "disease" circuits built into false animals; the construct which he now held on his lap had been put together in such a fashion that when a primary component misfired, the whole thing appeared-not broken-but organically ill.
> *Philip K. Dick, Do Androids Dream of Electric Sheep? (1968)*

Mau syntax contains several components. Some of these can be found on a single line of a page, while some can be found inside a block of text. In the previous section we already saw that a **paragraph** is one of the page components, but the full list is

- **Horizontal rule**: a line that contains three hyphens
- **Single line comment**: a line preceded by the characters `//`
- **Multi line comment**: any block of lines surrounded by the comment fence `////`
- **Variable definition**: a line that begins with a name surrounded by colons, e.g. `:var_name:`
- **Command**: a line that begins with a command name in the form `::command_name:`
- **Title**: a line that begins with a dot, e.g. `.Title`
- **Arguments**: a line that is completely surrounded by square brackets, e.g. `[arguments]`
- **Header**: a line preceded by one or more symbols `=`
- **Block**: any group of lines surrounded by a block fence of four identical characters, e.g. `++++`

- **Content**: one or more URIs preceded by `<<`
- **List**: a sequence of lines each one of them preceded by one or more symbols `#` or `*`
- **Paragraph**: any block of text preceded by an empty line

Inside paragraphs you can instead have the following special syntax

- **Styled text**: any sequence of words surrounded by `_underscores_`, `*stars*`, `~tildes~`, or `^carets^`
- **Verbatim text**: any sequence of words surrounded by backticks
- **Macros**: functions that come in the form `[name](arguments)`

Finally, there are some components that are created automatically, such as callouts and list entries, which will be introduced together with other components.

Components and rendering

As in other markup languages like Markdown, each Mau component has a default rendering. For example, the syntax `_some text_` will be rendered in HTML as `<em>some text</em>`, but the real power of Mau is that any component can be connected to a custom template that changes the rendering. For example, I might decide that `_some text_` should be rendered as `<span class="red">SOME TEXT</span>`.

Moreover, components can use different templates based on arguments or on the parent node that contains them. For example, I might decide that `_some text_` should be rendered as `<em>some text</em>` in normal paragraphs and as `<span class="red">SOME TEXT</span>` in a quote.

The template engine will be explained in depth in the second part of the documentation.

Arguments

Some Mau components 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 component.

- Macros receive arguments between round brackets: e.g. `[macro](argument1, argument2, ...)`.
- Commands receive some arguments directly after the final colon: e.g. `::command:argument1, argument2, ...`.
- Horizontal rules, commands, headers, blocks, content, lists, and paragraphs receive arguments on a separate line, surrounded by square brackets, e.g. `[argument1, argument2, ...]`.

In the next chapters there will be plenty of examples to show how to pass arguments and to clarify why commands can receive them in two different ways.

Named and unnamed arguments

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.

Unnamed arguments can never follow named ones, so the syntax `language=html, source` is invalid.

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 arguments are surrounded by round or square bracket, you have to use quotes whenever the value contains the closing bracket. E.g. `(attribution="The (real) author")`. 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`. Tags are useful to categorise elements and apply special formatting.

For example the arguments list `source, #custom, red, language=html` will internally be parsed as

```
Unnamed arguments: source, red
Named arguments: language=html
Tags: custom
```

Please note that the parsed name of the tag doesn't contain the hash mark `#`.

Subtype

The first unnamed argument whose value starts with `*` is interpreted as the subtype of the component and plays an important role in the choice of the rendering template, as will be clarified later in the documentation. For example, the syntax `*ctype1, arg1, arg2` will be parsed as

```
Unnamed arguments: arg1, arg2
Named arguments:
Tags:
Subtype: ctype1
```

You don't need to specify the subtype, but you should have maximum one of them. If you specify multiple ones Mau will at the moment keep only the first one. For example, the syntax `[*ctype1, *ctpe2, arg1, arg2]` will be parsed as the previous one and subtype `ctype2` will be lost.

Arguments and rendering

When a component is parsed, it is transformed into a node that is then passed to the rendering engine. Each node, whatever the component, has its arguments stored in the attributes `args`, `kwargs`, `tags`, and `subtypes`. See Basic Templates and List of Components for more information.

Chapter 3

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 `_underscores_`, `*stars*`, `~tildes~`, `^carets^`, 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.
```

Strategic **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 `if`s 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 using the macro `class`. 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 specific.

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 rule

You can add a separator or horizontal rule using three dashes `---` on a separate line.

Mau source

```
---
```

Chapter 4

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. In Mau, links are created with the macro `link`.

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 (see Mau Components).

Add text to the link

You can add 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.

Rich text

The text added to a link is rendered using Mau, so you can include styles or other macros, as long as the output makes sense in the target format. To avoid clashes between Mau syntax and arguments syntax, if you have rich text it is recommended to explicitly mark it as a named `text` argument.

Mau source

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

The source code can be found on **GitHub**.

In this case, the syntax `[link]("https://github.com/Project-Mau/mau", "*GitHub*")` would be interpreted as a link with subtype `GitHub` because of the leading `*`.

As for rich text, remember that Mau doesn't understand output formats, it merely uses templates. So, while it is possible in Mau to insert a link as a text for a link, that doesn't make sense in HTML or TeX.

Chapter 5

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
```

Mau converts it into a suitable header in the target format and 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
```

Table of Contents

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

Mau source

```
= Main section

Text of the main section

== Secondary section

Text of the secondary section

== Another secondary section

Text of another secondary section

=== A very specific section

Text of the very specific section

---

::toc:
```

This will be rendered as

You can avoid including specific headers in the TOC using tags. You can tag the headers you don't want to include and pass them to the command `::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

---

[exclude_tag=notoc]
::toc:
```

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

Headers and formatting

Headers can contain rich text just like paragraphs. For example you can have styles

Mau source

```
== Section _42_

== The function `answer()`
```

Anchors

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

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 configuration variable `mau.parser.header_anchor_function`

The function should accepts `text` and `level`, which are the text of the header and the number of equal signs preceding it, and return the ID as a string.

For example you can do something like

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

config = {
    "parser": {
        "header_anchor_function": custom_header_anchor
    }
}
```

You can read more about configuration variables in Variables and Configuration.

Header macro and IDs

Headers can be linked internally giving them an ID and using the macro `header`.

To assign an ID to a header simply use arguments

Mau source

```
[id=section-42]
= Section 42
```

Then, at any point of the document you can use the syntax `[header](section-42)` to generate a link to that header.

The text of the link is automatically set to the text of the header, but if you want you can provide a replacement, e.g. `[header](section-42, "the relative section")`.

Chapter 6

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 document 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 without a value are simply empty strings.

Mau source

```
:answer:

The answer is {answer}.
```

The answer is .

Variables can also be used in block definitions, see Blocks.

Preventing replacement

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 can have boolean values and be used as flags in conditional rendering (see Blocks). To create a boolean you need to use a `+` or `-` in front of the variable name, which will assig respectively a true or false value. The name of the variable will not contain the initial symbol.

```
:+trueflag:
:-falseflag:
```

When printed, boolean flags are empty strings.

Mau source

```
:+trueflag:
:-falseflag:

The first flag is {trueflag}. The second flag is {falseflag}.
```

The first flag is . The second flag is .

Conditional macro

Boolean variables are extremely useful for conditional blocks and conditional macros.

A paragraph can conditionally render some text using the macro `[if](variable, value_if_true, value_if_false)`. If the value of `variable` is true the text will be the rendering of `value_if_true`, otherwise it will be `value_if_false`.

Mau source

```
:+trueflag:
:-falseflag:

The value of the true flag is [if](trueflag, TRUE, FALSE).

The value of the false flag is [if](falseflag, TRUE, FALSE).
```

The value of the true flag is TRUE.

The value of the false flag is FALSE.

As always with macros, `value_if_true` and `value_if_false` can contain rich text.

Mau source

```
:+trueflag:

The value of the true flag is [if](trueflag, true="*TRUE*",
↪  false="_FALSE_").
```

The value of the true flag is **TRUE**.

If you need to conditionally render big chunks of text, however, you should consider using conditional blocks.

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 Configuration).

Chapter 7

Lists

- “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)

Unordered 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

Ordered lists

You can 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

Mixed lists

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

Starting index

For ordered lists, the default starting index is 1, but this can be changed setting an attribute.

Mau source

```
[start=42]
# Step 1
# Step 2
## Step 2a
## Step 2b
# Step 3
```

42. Step 1
43. Step 2
    a) Step 2a
    b) Step 2b
44. Step 3

Sometimes, it's useful to have lists split in multiple sections, and have each section continue automatically from the previous one. This can be done setting `start=auto`

Mau source

```
# Step 1
# Step 2

This is a paragraph that interrupts the list

[start=auto]
# Step 3
# Step 4

This is another paragraph that interrupts the list

[start=auto]
# Step 5
# Step 6
```

1. Step 1
2. Step 2

This is a paragraph that interrupts the list

1. Step 3
2. Step 4

This is another paragraph that interrupts the list

1. Step 5
2. Step 6

Chapter 8

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 syntax `<< image:URI`

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
```

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
```

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 9

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 `[*source]` and surrounding the code with a fence of four hyphens `----`

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 immediately after `*source`

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
```

Highlighting for TeX output format is not provided by default inside Mau. Templates make it easy to use third-party tools, however. For example, this book has been highlighted using minted.

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
----
```

Titles

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

Mau source

```
. Header anchor function
[*source]
----
def header_anchor(text, level):
    """
    A simple Python function
    """

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

Chapter 10

Footnotes

> 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).

Footnotes

The text of a footnote is defined in a way similar to code blocks, using `[*footnote, NAME]` and a fence of four hyphens `----`. `NAME` is the name of the footnote, that has to be unique in the document.

Mau source

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

The text between the fences is parsed by Mau as a stand-alone document, which means that you can use the syntax we discussed so far.

Mau source

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

You can create a reference to the footnote with the macro `[footnote](NAME)`, which is usually rendered as a superscript small number.

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 ---
13
14 ::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 Basic Templates).

The definition of a footnote can appear before or after the reference in the text. Both blocks and macros are collected by the parser and connected at the end of the process.

Chapter 11

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)

Basic syntax

Blocks, are groups of lines surrounded by a block fence of four identical characters

Mau source

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

The content of the block is parsed by Mau exactly like the rest of the document, unless the block is configured in a different way. As blocks are rendered with specific templates, they are useful to create alerts, asides, and in general custom rendering of blocks of text.

As you already saw in the previous chapters, the block syntax is also used by Mau to define some components like code blocks, footnotes, and references.

Fences

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.
```

Nested blocks

Blocks can contain other blocks, and Mau doesn't have any upper limit to the number of nesting levels. The only requirement is that you need to use a different character for the inner fence.

Mau source

```
++++
This is a block

%%%%
This is another block inside the previous one
%%%%
++++
```

Titles

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

Mau source

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

The space between the dot and the title is optional

Mau source

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

Attributes

You can pass arguments to blocks with the syntax `[arguments]` on a line before the opening fence

Mau source

```
[*btype1, arg1, arg2, #tag1, key1=value1]
----
This block has

* Subtype: `btype1`
* Unnamed arguments: `["arg1", "arg2"]`
* Tags: `["tag1"]`
* Named arguments: `{"key1": "value1"}`.
----
```

You can specify title and attributes in any order.

Mau source

```
. Title
[arguments]
----
Block content
----

[arguments]
. Title
----
Block content
----
```

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

Mau source

```
. Title

[arguments]

----
Block content
----
```

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 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
```

Chapter 12

Built-in Blocks 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
```

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

Admonitions

Admonitions are blocks 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, "fas fa-info", "Info"]
----
This is my note
----
```

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 with the syntax `condition=CONDITION:VARIABLE:[VALUE]`.

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

Boolean variables

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

```
:+render:

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

:-render:

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

This will be rendered

Reverse condition

You can reverse the condition using `ifnot`

```
:+render:

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

Condition and subtype

Conditions can be used together with other arguments, in particular the subtype.

```
:+detailed:

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

[*aside, 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.

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

Including content

As mentioned in 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 13

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)

Block engines

Mau blocks have an advanced feature called `engine`, which rules the way Mau processes the content of the block.

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`
- `footnote`
- `reference`
- `raw`
- `mau`

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.

We saw examples of this engine when we discussed quotes and admonitions.

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.

The block subtype `source` that we saw in Code Blocks is a shortcut notation to use the engine `source`. The notation `[*source]` is equivalent to `[engine=source]`.

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 Basic 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`, `[*footnote]` is translated into `[engine=footnotes]` and `[*reference]` into `[engine=references]`.

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, you might want to add custom HTML code.

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 Basic Templates 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

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

== Secondary 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.

Custom block definitions

As we saw previously, blocks have many attributes that you can set on them. In Basic Templates and Advanced 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:ALIAS, ARGUMENTS`.

Mau source

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

[*python]
----
class MyException(Exception):
    pass
----
```

This is equivalent to

Mau source

```
[engine=source, language=python]
----
class MyException(Exception):
    pass
----
```

Mandatory arguments and defaults

The full syntax of a block definition is

Mau source

```
::defblock:ALIAS, [*SUBTYPE], [UNNAMED ARGUMENTS], [NAMED ARGUMENTS]
```

but the aliasing mechanism doesn't work as a pure substitution.

- `SUBTYPE` is going to be the block subtype. If missing, the subtype will not be set on the block.
- `UNNAMED ARGUMENTS` are interpreted as **mandatory arguments**.
- `NAMED ARGUMENTS` are interpreted as **defaults values**.

For example, `::defblock:alias, *type1, name` means that whenever we use `alias` as the block subtype we also have to provide either an unnamed argument (which will be assigned the key `name`) or a named argument called `name`.

Mau source

```
::defblock:alias, arg1, key1=value1

[*alias]
----
This is invalid, as it is missing the unnamed argument
----

[*alias, alert]
----
This is valid.
Unnamed arguments: []
Named arguments: {"arg1": "alert", "key1": "value1"}.
----

[*alias, alert, red]
----
This is valid.
Unnamed arguments: ["red"]
Named arguments: {"arg1": "alert", "key1": "value1"}.
----

[*alias, alert, red, key1=anothervalue]
----
This is valid.
Unnamed arguments: ["red"]
Named arguments: {"arg1": "alert", "key1": "anothervalue"}.
----

[*alias, red, arg1=alert]
----
This is valid.
Unnamed arguments: ["red"]
Named arguments: {"arg1": "alert", "key1": "anothervalue"}.
----
```

As you can see, block definitions do not work exactly like functions in programming languages. Block attributes can always contain any number of unnamed and named arguments, and the block definition just specifies which **named ones** we are always expecting to see.

Recursive subtypes

Block definitions are not recursive, so it is perfectly fine to write a definition like

Mau source

```
::defblock:alias, *alias, engine=raw, classes="myclass1,myclass2"
```

In this case the syntax `[*alias]` will be converted to `[*alias, engine=raw, classes="myclass1,myclass2]` and the block will still have the subtype `alias`.

Predefined blocks

As we saw in the previous sections, Mau provides some aliases out of the box, namely `source`, `footnote`, `reference`, and `admonition`. While these are created directly in the Python soruce code, they are equivalent to the following definitions

Mau source

```
::defblock:source, language=text, engine=source
::defblock:footnote, name, engine=footnote
::defblock:reference, 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.

Block definitions in configuration

You can create block definitions through Mau's configuration setting the variable `mau.parser.block_definitic`

Mau source

```
"mau" : {
    "parser": {
        "block_definitions": {
            "alias": {
                "subtype": "type",
                "mandatory_args": ["arg1", "arg2"],
                "defaults": {"key1": "value1"},
            },
        },
    }
}
```

Chapter 14

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 *important* text.
```

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

Abstract Syntax Tree

```yaml
data:
  type: document
  content:
  - data:
      type: paragraph
      content:
      - data:
          type: text
          value: 'Stars identify '
      - data:
          type: style
          value: star
          content:
          - data:
              type: text
              value: important
      - data:
          type: text
          value: ' text.'
```

You can the AST at any time using the output format `yaml` on the command line

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

Once the AST has been created, the chosen visitor is used to 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

```yaml
- data:
    type: style
    value: star
    content:
    - data:
        type: text
        value: important
```

Which involves the template `text.html` and `style.star.html`

text.html

```
{{ value }}
```

style.star.html

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

The output of the style node will then be

HTML output

```
<strong>important</strong>.
```

And the output of the whole text will be

HTML output

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

If we changed the template to

style.star.html

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

the result would be

HTML output

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

Jinja

Templates are currently implemented using the Jinja templating engine. This means that inside templates you can use filters, statements, loops, and everything that Jinja provides.

In particular, you can leverage commands like `extends` and `include` to avoid repeating yourself and to tidy up the structure of your templates.

Define templates

Mau templates can be provided in the YAML configuration or as individual files.

YAML Configuration file

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

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

The file has to be named after the template and have an extension matching the output format (see the paragraph about visitors). 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.

Template files can be created with a flat or nested structure. Each file can be created with its full name, so you can have

```
templates/
  header.html
  source.html
  source.python.html
  source.javascript.html
  style.star.html
  style.underscore.html
```

or you can use each part of the dotted name as a subdirectory

```
templates/
  header.html
  source.html
  source/
    python.html
    javascript.html
  style/
    star.html
    underscore.html
```

This is true at individual level, so you can mix the two styles

```
templates/
  header.html
  source.html
  source.python.html
  source/
    javascript.html
  style/
    star.html
    underscore.html
```

Visitor plugins

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

At the moment, there are also two plugins that provide a subclass of the `JinjaVisitor` and specifically target HTML (`mau-html-visitor`) and TeX (`mau-tex-visitor`). They can be installed in any virtual environment using `pip`

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

The two plugins provide default templates for their output format. You can see them in the relative repositories

- Mau HTML visitor: https://github.com/Project-Mau/mau-html-visitor
- Mau TeX visitor: https://github.com/Project-Mau/mau-tex-visitor

Template plugins

Mau supports plugins that only add templates, such as `mau-docs-templates`. Such plugins provide only template files, without a specific visitor.

The templates provided by the plugin can be used directly as they are loaded by visitors, or can be included in your own templates using Jinja `extend` or `include`.

Chapter 15

Advanced Templates

"In some ways you're so advanced, and yet when it comes to making a decision, you're still a child. I can't decide for you, Charlie. The answer can't be found in books — or be solved by bringing it to other people."
Daniel Keyes, Flowers for Algernon (1959)

Examples of templates, using the parent, using tags.

Template names

Each component of a Mau document is transformed into a node of the AST, and each node is rendered using a template. For each node, Mau creates a list of several templates and tries to locate them. The first matching template will be loaded and used to render the node. Templates have a specific extension that depends on the output format (`.html` for HTML, `.tex` for TeX, and so on).

The names of possible templates are created considering several parts: the node templates, the node subtype, the node tags, the parent node, and the visitor prefixes, with the following schema:

```
{prefix}.{parent_type}.{parent_position}.{node_template}.{node_subtype}.{node_tag}
```

and the Python that creates the list of possible template names is

```
templates = [
    f"{prefix}{parent_type}{node_template}{node_subtype}{node_tag}"
    for prefix in prefixes
    for parent_type in parent_types
    for node_template in node_templates
    for node_subtype in node_subtypes
    for node_tag in node_tags
]
```

Node templates

When Mau parses the text `*important*`, it generates a node with type `style` and value `star`. Such a node has two possible templates

- `style.star`
- `style`

While the first is specific for the style `star`, the second one is shared with styles `underscore`, `caret`, and `tilde`. This means that for HTML output you can create the file `style.html` that serves all styles, and then the file `style.star.html` to customise that specific style. As Mau provides default templates, you can clearly define only the latter.

The list of templates used for a specific node depends on the type of node and will be documented in the relative section.

Node subtype

As some nodes can have a subtype the list of templates takes that into account. For each node template, Mau first tests if there is a more specific one that includes the subtype, so the list of templates is

- `{node_template1}.{node_subtype}`
- `{node_template1}`
- `{node_template2}.{node_subtype}`
- `{node_template2}`
- ...
- `{node_type}.{node_subtype}`

- `{node_type}`

where `node_template1`, `node_template2`, and so on are specific to the type of node.

For example, when Mau renders the text

This is an important paragraph

the list of possible templates is

- `paragraph.important`
- `paragraph`

Node tags

Each tag assigned to the node is used to find specialised templates appending the tag at the end of the template name, so the list is

- `{node_template1}.{tag1}`
- `{node_template1}.{tag2}`
- `{node_template1}`
- `{node_template2}.{tag1}`
- `{node_template2}.{tag2}`
- `{node_template2}`
- ...
- `{node_type}.{tag1}`
- `{node_type}.{tag2}`
- `{node_type}`

For example, when Mau renders the text

This is an important paragraph

the list of possible templates is

- `paragraph.tag1`
- `paragraph.tag2`
- `paragraph`

Parent node

Each node is connected with the node that contains it, so the template can be chosen depending on the parent node type, the parent node subtype, or the position that the node has in the parent. For each template `TEMPLATE` the following variants are tested

- `{parent.node_type}.{parent.subtype}.{parent_position}.TEMPLATE`
- `{parent.node_type}.{parent.subtype}.TEMPLATE`
- `{parent.node_type}.{parent_position}.TEMPLATE`
- `{parent.node_type}.TEMPLATE`
- `TEMPLATE`

The value `parent_position` is currently used only for blocks, and can be `primary` for primary content and `secondary` for secondary content.

For example, consider the following document. A paragraph with subtype `important` as primary content of a block with subtype `aside`

This is an important paragraph

the list of possible templates is

- `block.aside.primary.paragraph.important`
- `block.aside.primary.paragraph`
- `block.aside.paragraph.important`
- `block.aside.paragraph`
- `block.primary.paragraph.important`
- `block.primary.paragraph`
- `block.paragraph.important`

- `block.paragraph`
- `paragraph.important`
- `paragraph`

Visitor prefixes

When Mau renders a document it can be assigned a list of prefixes, used to identify templates. This is useful when the same parsed material has to be rendered in different ways at the same time. For example, the table of contents might be rendered using certain templates in the main area of an HTML page, but using different ones in the navigation bar.

The list of all templates is tested using each one of the prefixes, including the empty one. For example, if Mau is given the prefixes `prefix1` and `prefix2` the document

This is an important paragraph

corresponds to the templates

- `prefix1.block.aside.primary.paragraph.important`
- `prefix1.block.aside.primary.paragraph`
- `prefix1.block.aside.paragraph.important`
- ...
- `prefix1.paragraph`
- `prefix2.block.aside.primary.paragraph.important`
- `prefix2.block.aside.primary.paragraph`
- `prefix2.block.aside.paragraph.important`
- ...
- `prefix2.paragraph`
- `block.aside.primary.paragraph.important`
- `block.aside.primary.paragraph`
- `block.aside.paragraph.important`
- ...
- `paragraph`

Chapter 16

List of Components

The financial records were fairly simple — the purchase of this or that trivial toy or bit of trumpery jewelry; lists of presents given and received; a somewhat more meticulous listing of jewels of genuine value, inheritances, or gifts.
Lois McMaster Bujold, The Curse of Chalion (2001)

This chapter contains the full documentation of all Mau components. For each one of them you will find:

- **Description**: a simple description of the component.
- **Scope**: whether the node can be found in a `paragraph` or in a `document`.
- **See**: a list of related nodes.
- **Syntax**: a description of the syntax used to create the component.
- **Templates**: the node templates for this component. See Basic Templates to learn about templates.
- **Fields**: the variables passed to the Jinja template.

Fields have a name, a type, and a description. Type is one of

- `bool`: a Python boolean that can be used directly in Jinja conditional statements.
- `int`: a Python integer.

- `list[type]`: a list of values of the given type.
- `str`: a Python string that contains pure text.
- `TARGET`: text in 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.

Every node receives the following fields, which won't be repeated for each component

- `parent: NODE = None` The parent node.
- `parent_position: primary | secondary = None` The position of the node in the parent.
- `children: list[NODE] = []` The list of children nodes.
- `subtype: str = None` The subtype of the node.
- `args: list[str] = []` The unnamed arguments.
- `kwargs: dict[str] = {}` The named arguments.
- `tags: list[str] = []` The tags.

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

Not all components in this list can be directly created in Mau documents. Some, like callouts or ToC entries, are created automatically as part of other components. However, such nodes will be present in the AST, and thus can be styled using templates.

Block

Description:

This node represents a generic block. Blocks can isolate text and process it in several different ways depending on the `engine`. Blocks capture the text between fences into `primary_content` and the paragraph immediately after the closing fence (adjacent to it) into `secondary_content`.

The block syntax is shared with more specific components like Source (block), Footnote (block).

Documentation: Blocks

Scope: `document`

See: Source (block), Footnote (block).

Syntax:

Mau source

```
. Optional title
[OPTIONAL ARGUMENTS]
----
CONTENT
----
```

Fences are made of any 4 identical characters.

Templates:

- `block.{engine}`
- `block`

Fields

- `classes: list[string] = []` A list of classes assigned to this block.
- `title: str = None` A title specified before this block.
- `engine: str = None` The engine that Mau used to process the block.
- `preprocessor: str = None` The preprocessor that Mau used for this block.
- `content: TARGET`: the primary content (Mau text between fences).

- `secondary_content: TARGET`: the secondary content (Mau text just below the closing fence.

Callout

Description:

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

This node cannot be created in isolation, it is generated automatically by source blocks.

Documentation: Code Blocks

See: Source (block)

Templates

- `callout`

Fields

- `marker: str` The label of this callout.

Callout entry

Description:

This node renders an entry 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.

This node cannot be created in isolation, it is generated automatically by source blocks.

Documentation: Code Blocks

See: Source (block)

Templates

- `callouts_entry`

Fields

- `marker: str` The label of this callout.
- `value: str` The text associated with this callout.

Caret (style)

Description:

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

Documentation: Text Formatting

See: Star (style), Tilde (style), Underscore (style)

Scope: `paragraph`

Syntax `^TEXT^`

Templates

- `style.caret`

Fields

- `value: str = caret` The style type.
- `content: TARGET` The content of the node.

Class (macro)

Description:

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

Documentation: Text Formatting

Scope: `paragraph`

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

Templates

- `macro.class`

Fields

- `classes: list(str)` The list of classes.
- `content: TARGET` The content of the node.

Container

Description:

This node is used to wrap the whole document and provide a wrapper template. This is the default value of the variable `mau.parser.content_wrapper`.

Scope: `document`

Templates

- `container`

Fields

- `content: TARGET` The content of the container.

Content

Description:

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

See: Image (content)

Scope:`document`

Syntax:

Mau source

```
. Optional title
[OPTIONAL ARGUMENTS]
<< TYPE:URI
```

Templates

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

Fields

- `content_type:` `str` The type of this content.
- `title:` `TARGET` An optional title for the content.

Document

Description:

This node is used to wrap the whole document and provide a wrapper template. You can use this node passing `DocumentNode` through the variable `mau.parser.content_wrapper`.

Scope: `document`

Templates

- `document`

Fields

- `content:` `TARGET` The content of the document.

Footnote (block)

Description:

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 mention (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 node. Since LaTeX manages footnotes automatically, the default template in that case is empty.

Documentation: Footnotes

See: Footnote (macro), Footnotes (command)

Scope: `document`

Syntax

Mau source

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

Templates

- `footnotes_entry`

Fields

- `content:` `TARGET` The content of the footnote.
- `number:` `str` The number of this footnote.
- `reference_anchor:` `str` The anchor to the footnote mention.
- `definition_anchor:` `str` The anchor to the footnote definition.

Footnote (macro)

Description:

This node defines the mention to a footnote. It has to be paired with a block `footnote` (see Footnote (block)) 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 mention 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.

Documentation: Footnotes

See: Footnote (block), Footnotes (command)

Scope: `paragraph`

Syntax: `[footnote](NAME)`

Templates

- `macro.footnote`

Fields

- `content: TARGET` The content of the footnote defined by the companion block.
- `number: str` The number of this footnote.
- `reference_anchor: str` The anchor to the mention.
- `definition_anchor: str` The anchor to the definition.

Footnotes (command)

Description:

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.

Documentation: Footnotes

See: Footnote (block), Footnote (macro)

Scope: `document`

Syntax:

```
[OPTIONAL ARGUMENTS]
:
```

Templates

- `footnotes`

Fields

- `entries: list(TARGET)` This is the list of footnotes, each being the rendering of a footnote entry.

Header

Description:

This node renders a header. Each header is automatically stored in the ToC. A header can be given an `id` that is then linked using a macro `header`.

Headers are given an anchor created using a default function. The function can be replaced using the variable `mau.parser.header_anchor_function`, and the individual anchor of a header can be overwritten with the keyword `anchor`.

Documentation: Headers

See: ToC (command), ToC entry, Header (macro)

Scope: `document`

Syntax:

Mau source

```
[OPTIONAL ARGUMENTS]
= Header
```

Templates

- `header`

Fields

- `value: TARGET` The text of the header.
- `level: int` The level of the header, 1 being the highest.
- `anchor: str` The anchor of the header if the template wants to set up cross-references between it and the ToC.

Header (macro)

Description:

This node represents an link to a header in the same document. The macro requires the ID of a header, and accepts an optional text. If the text is not provided, the text of the linked header will be used.

Documentation: Headers

See: Header

Scope: `paragraph`

Syntax: `[header](ID, TEXT)`

Templates

- `macro.header`

Fields

- `header: dict`
  - `anchor: str` The anchor of the connected header.
  - `value: TARGET` The text of the header.
  - `level: int` The level of the header, 1 being the highest.
- `content: TARGET` The text of the link if specified, otherwise the text of the header.

Horizontal rule

Description:

This node inserts a horizontal rule.

Scope: `document`

Syntax:

Mau source

```
[OPTIONAL ARGUMENTS]
---
```

Templates

- `horizontal_rule`

Image (content)

Description:

This node inserts an image in the document. The image is inserted as a document node, if you want to include images in paragraphs use the macro `image`.

You can pass alternate text through the keyword `alt_text` and classes through `classes`.

Documentation: Images

See: Content, Image (macro)

Scope: `document`

Syntax:

Mau source

```
. Optional title
[OPTIONAL ARGUMENTS]
<< image:URI
```

Templates

- `content_image`

Fields

- `uri:` `str` The URI of the image file.
- `alt_text:` `str` Alternative text to show if the image cannot be loaded.
- `classes:` `list[str]` List of classes to append to the target node.
- `title:` `str` Title (caption) of the image.

Image (macro)

Description:

Through this node, Mau inserts an image in a paragraph. The macro accepts alternate text, width and height.

Documentation: Images

Scope: `paragraph`

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

Templates

- `macro.image`

Fields

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

List

Description:

This node 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).

This node is created automatically when a list item is found in the document. The node is also used for nested list items.

Documentation: Lists

See: List item

Syntax:

Mau source

```
[OPTIONAL ARGUMENTS]
* List items
```

Templates

- `list`

Fields

- `ordered: bool` Whether the list is ordered or not.
- `items: TARGET` The list items.
- `main_node: bool` Whether this is the main list of a sublist.

List item

Description:

This node renders a single item in a list.

Documentation: Lists

See: List

Scope: `document`

Syntax:

Mau source

```
* Unordered node level 1
** Unordered node level 2
...
# Ordered node level 1
## Ordered node level 2
```

A list item is always contained in a list node.

Templates

- `list_item`

Fields

- `level:` `int` The level of nesting (1 is the first level).
- `content:` `TARGET` The content of this item (which might be a sublist).

Link (macro)

Description:

This node represents a hypermedia link to a URL. The macro accepts a `target` (a URL) and optional `text`. If the text is not provided, the target is used instead.

Documentation: Links

See: Mailto (macro)

Scope: `paragraph`

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

Templates

- `macro.link`

Fields

- `target: str` The target URL.
- `text: TARGET` The text of the link.

Mailto (macro)

Description:

This node represents an email link. The macro is a wrapper around Link (macro), but the output URI is prefixed by `mailto:`.

Documentation: Links

Paragraph

Description:

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

Documentation: Basic Syntax

Scope: `document`

Syntax:

Mau source

```
. Optional title
[OPTIONAL ARGUMENTS]
Any line of text that doesn't match another page node.
```

Templates

- `paragraph`

Fields

- `content: TARGET`: the content of the paragraph

Source (block)

Description:

This node 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.

You can pass the keywords `callouts`, `highlights`, and `language`.

There are three nodes rendered by templates: the block itself, the callout label attached to a line, and each 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`.

Documentation: Code Blocks

See: Block, Callout, Callout entry

Scope: `document`

Syntax

Mau source

```
[*source, LANGUAGE]
----
CONTENT
----
```

Templates

- `source.{language}`
- `source`

Fields

- `language: str` The language of the code contained in this block
- `callouts: list[TARGET]` A list of callout entries
- `markers: list[SPECIAL]` List of markers (see description)
- `highlights: list[int]` List of lines that have to be highlighted
- `delimiter: str` Callouts delimiter
- `code: SPECIAL` Code and callouts (see description)
- `title: str` Title of this block
- `classes: list[str]` A list of classes assigned to this block
- `preprocessor: str` The preprocessor (currently not in use)

Star (style)

Description:

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

Documentation: Text Formatting

See: Caret (style), Tilde (style), Underscore (style)

Scope: `paragraph`

Syntax `^TEXT^`

Templates

- `style.star`

Fields

- `value: str = star` The style type.
- `content: TARGET` The content of the node.

Tilde (style)

Description:

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

Documentation: Text Formatting

See: Caret (style), Star (style), Underscore (style)

Scope: `paragraph`

Syntax: `~TEXT~`

Templates

- `style.tilde`

Fields

- `value: str = tilde` The style type.
- `content: TARGET` The content of the node

ToC (command)

Description:

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

Documentation: Headers

See: Header, ToC entry

Scope: `document`

Syntax:

Mau source

```
[OPTIONAL ARGUMENTS]
:
```

Templates

- `toc`

Fields

- `entries: list[TARGET]` This is the list of headers, each being the rendering of a ToC entry.

ToC entry

Description:

This node renders a node in the ToC.

Documentation: Headers

See: Header, ToC (command)

Templates

- `toc_entry`

Fields

- `value: str` The header contained in this node
- `anchor: str` The anchor of the header
- `children: TARGET` The rendered list of children (rendered as the whole ToC)

Underscore (style)

Description:

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

Documentation: Text Formatting

See: Caret (style), Star (style), Tilde (style)

Scope: `paragraph`

Syntax: `_TEXT_`

Templates

- `style.underscore`

Fields

- `value: str = underscore` The style type.
- `content: TARGET` The content of the node.

Verbatim

Description:

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

Documentation: Text Formatting

Scope: `paragraph`

Syntax: `‘TEXT‘`

Templates

- `verbatim`

Fields

- `value:` `str` The verbatim text

Chapter 17

Python Interface

"If you have any questions, your PDA can port into the Henry Hudson information system and use the AI interface to assist you; just use your stylus to write the question or speak it into your PDA's microphone."
John Scalzi, Old Man's War (2005)

Mau can be used inside any Python project through the object `Mau`. The following sketches the steps you need to go through. The best way to see that in action is in the code of the Pelican Mau reader plugin.

```python
from mau import Mau, load_visitors
from mau.environment.environment import Environment
from mau.errors import MauErrorException, print_error

visitor_classes = load_visitors()
visitors = {i.format_code: i for i in visitor_classes}

self.environment = Environment()

config = {} # This is a Python dictionary containing the configuration
output_format = "html" # This depends on the output that you want to
↪  create

if output_format not in visitors:
    raise OutputFormatNotSupported(output_format)

visitor_class = visitors[output_format]
self.environment.setvar("mau.visitor.class", visitor_class)
self.environment.setvar("mau.visitor.format", output_format)

mau = Mau(
    source_path, # The source file
    self.environment,
)

try:
    with pelican_open(source_path) as text:
        mau.run_lexer(text)

    mau.run_parser(mau.lexer.tokens)

    content = mau.run_visitor(mau.parser.output["content"])

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

except MauErrorException as exception:
    ... # Manage exception
```

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
```

See Variables to learn more about Mau variables.

Chapter 18

Configuration

Roger, of course, was not a normal human being. His muscular system was content with prolonged overloads in almost any configuration it could bend into at all.
Frederik Pohl, Man Plus (1976)

Currently, Mau supports the following configuration variables

mau.parser

`mau.parser.block_definitions`

Custom block definitions that we want to create through configuration instead of using `::def-block:`.

`mau.parser.header_anchor_function`

A function that creates a unique anchor for headers. The prototype is `def header_anchor(text:str, level:str) -> str`, where `text` is the content of the header and `level` is the depth level.

`mau.parser.content_wrapper`

The node to use to wrap the content. The default value is `ContainerNode`.

mau.visitor

`mau.visitor.class`

The Python class that implements the visitor interface and provided by a plugin.

`mau.visitor.prefixes`

The template prefixes that should be used.

`mau.visitor.template_providers`

The template providers plugin we want to load. Mau autodetects template plugins but doesn't automatically load them. They need to be explicitly activated.

`mau.visitor.templates_directory`

The directory that contains template files, saved either with a flat or a nested structure. See Basic Templates.

`mau.visitor.custom_templates`

Custom templates that we want to define through configuration. See Basic Templates.

History of Mau

"I am not certain, Master Ladrian," Sazed said. "However, understanding the real history behind the Ascension will be of use, I think. At the very least, it will give us some insight to the Lord Ruler's mind."
Brandon Sanderson, The Final Empire (2006)

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!