Docco: Feature Showcase

Welcome to this demonstration of Docco's capabilities.

1 Key Features

This document demonstrates Docco's core capabilities:

- Inline file inclusion with placeholder substitution
- Table of contents generation
- Page layout directives (page breaks and orientation changes)
- Headers and footers with directive support
- Multi-language support
- Python code execution

1.1 Frontmatter

YAML frontmatter at the beginning of the document (between `---` delimiters) configures document processing:

```
---
css:
  - "css/page.css"
  - "css/toc.css"
multilingual: true
base_language: en
---
```

1.1.1 Supported Fields

`css` - CSS stylesheet(s) for PDF styling. Can be:

- Single file (string): `css: "style.css"`
- Multiple files (inline array): `css: ["page.css", "theme.css"]`
- Multiple files (multiline list):

```
css:
  - "css/page.css"
  - "css/toc.css"
```

Paths are relative to the markdown file. Additional CSS can also be provided via the CLI `--css` argument, which overrides frontmatter styles.

`multilingual` - Enable multilingual mode (boolean, default: `false`). When set to `true`, Docco automatically extracts translatable strings to a POT file and generates PDFs for the base language plus all discovered translations.

base_language - The language code of the source document (required when `multilingual: true`). Example: `base_language: en`. This will be used as the suffix for the base language PDF (e.g., `Document_EN.pdf`).

header and **footer** - HTML files for page headers and footers with placeholder and directive support. See the [Headers & Footers](#) section for details.

1.2 Multilingual Support

Docco supports creating professional multilingual documents with automatic language-specific PDF generation. There are two approaches:

1.2.1 Approach 1: Multilingual Mode (Recommended)

Use the `multilingual: true` flag in frontmatter along with `base_language` to automatically generate PDFs for the base language plus all available translations:

```
---
multilingual: true
base_language: en
---
```

When enabled, Docco will:

1. Extract a POT file to a `{document_name}/` subfolder
2. Discover all `.po` translation files in that subfolder
3. Generate a PDF for the base language (e.g., `Document_EN.pdf`)
4. Generate translated PDFs for each `.po` file (e.g., `Document_DE.pdf`, `Document_FR.pdf`)

Example: See `Multilingual_Document_Example.md` which generates `Multilingual_Document_Example_EN.pdf`, `Multilingual_Document_Example_DE.pdf`, and `Multilingual_Document_Example_NL.pdf`.

1.2.2 Approach 2: Manual Translation Workflow

For single-language builds with specific translations, use the `--po` flag:

```
docco myfile.md --po translations/de.po -o output/
```

1.3 Professional Translation Workflow with POT/PO Files

Docco integrates with professional translation tools and services:

1.3.1 Step 1: Extract Translatable Strings

```
docco extract myfile.md -o translations/
```

This generates a `myfile.pot` file containing all translatable strings from your markdown.

1.3.2 Step 2: Create Language-Specific Translations

Translators create `.po` files for each language using professional tools:

- **poedit** (desktop application)
- **Weblate** (web-based, collaborative)
- **Crowdin, Lokalise, POEditor** (professional translation platforms)
- Any gettext-compatible tool

File structure:

```
myfile.md
myfile/
    ├── myfile.pot        (template)
    ├── de.po             (German translation)
    ├── fr.po             (French translation)
    └── nl.po             (Dutch translation)
```

1.3.3 Step 3: Generate Multilingual PDFs

With `multilingual: true` in frontmatter:

```
docco myfile.md -o output/
```

This generates one PDF per language automatically.

Or manually for specific translations:

```
docco myfile.md --po translations/de.po -o output/de/
```

1.3.4 Translation Maintenance

When the source document changes:

```
# Re-extract POT
docco extract myfile.md -o translations/

# Merge with existing translations (updates only new/changed strings)
msgmerge -U translations/de.po translations/myfile.pot
```

This allows translators to focus on new content rather than re-translating the entire document.

1.4 General Directive Rule

Directives can appear anywhere in the document, including in the middle of lines. However, directives inside code blocks (both inline `code` and fenced blocks) are **protected** and will not be processed. This allows you to show directive syntax as examples in documentation without triggering them.

1.5 Inlined Content

The `inline` directive embeds external markdown files with placeholder substitution.

Syntax: `<!-- inline:"foobar.md" key1="value1" key2="value2" -->`

1.5.1 Arguments

All attributes after the path are placeholders. For example, `author="Docco Team"` replaces `{{author}}` in the inlined file. Arguments are optional. Spaces around the `inline` keyword and colon are accepted. See example below:

This content is inlined, with arguments.

Author: Docco Team Date: 2025-10-26

1.5.2 Recursive Inlining

Inlined markdown files can themselves contain inline directives, allowing for multi-level composition (up to a maximum depth of 10 levels to prevent infinite recursion). This enables modular document structures where content can be composed from multiple nested files.

1.6 Table of Contents

The `<!-- TOC -->` directive generates a hierarchical table of contents with automatic numbering (1, 1.1, 1.2, etc.).

Use `<!-- toc:exclude -->` before a heading to exclude it from the TOC and remove its numbering.

1.7 Page Layout Directives

1.7.1 Page Breaks

This section starts on a **new page** using the `<!-- page break -->` directive.

1.7.2 Landscape Orientation

This section uses **landscape orientation** with the `<!-- landscape -->` directive, providing more horizontal space for wide content.

Q1 Revenue	Q1 Expenses	Q1 Profit	Q2 Revenue	Q2 Expenses	Q2 Profit	Q3 Revenue	Q3 Expenses	Q3 Profit	Q4 Revenue	Q4 Expenses
$50,000	$35,000	$15,000	$55,000	$37,000	$18,000	$62,000	$40,000	$22,000	$71,000	$45,000

1.7.3 Back to Portrait

This section returns to **portrait orientation** using the `<!-- portrait -->` directive.

2 Headers & Footers

Docco supports page headers and footers using HTML files with full directive support. Headers and footers are processed through the same pipeline as the main document, allowing for:

- Placeholder substitution
- Dynamic content with `<!-- python -->` directives
- File inclusion with `<!-- inline -->` directives

2.1 Configuration

Headers and footers are configured in the frontmatter:

```
header:
  file: "header.html"
  title: "Docco Feature Showcase"
  author: "Docco Team"
footer:
  file: "footer.html"
  title: "Docco"
```

The `file` key specifies the HTML file path. All other keys are placeholders that replace `{{key}}` in the HTML file.

2.2 CSS Requirements

Headers and footers require CSS to position them on the page using CSS Paged Media. Example:

```
@page {
    margin-top: 2.5cm;
    margin-bottom: 2.5cm;
    @top-center {
        content: element(header);
    }
}
```

```
    @bottom-center {
        content: element(footer);
    }
}
```

See `examples/css/header_footer.css` for a complete example.

2.3 Example Files

Header (`header_showcase.html`):

```
<div style="text-align: center; color: #666;">
    {{title}} - {{author}}
</div>
```

Footer (`footer_showcase.html`):

```
<div style="display: flex; justify-content: space-between;">
    <span>{{title}} - Generated: <!-- python -->
import datetime
print(datetime.datetime.now().strftime("%Y-%m-%d"))
    <!-- /python --></span>
    <span class="page-number"></span>
</div>
```

3 Python Code Execution

The `<!-- python -->` directive executes Python code and inserts the stdout output into the markdown. This is useful for generating dynamic content.

The output can contain other directives (markdown, inline files, etc.).

Important: Python code execution is disabled by default for security reasons. Use the `--allow-python` flag to enable it.

Syntax: `<!-- python -->`code`<!-- /python -->`

Example - a loop that outputs digits

This code:

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

```
    print(i, end='')
print("_", end='')
```

Returns: *0123456789*

4 Markdown examples

Docco relies on [MarkdownIt](#) for rendering markdown to HTML. It fully supports the [Commonmark specs](#) extend with table support. In addition, the [(block) attributes](#) plugin is installed.

4.1 Images with styling

Add images using the normal Markdown way, and use {} add style(s) to the images. For example:

```
![](images/idea.svg){.icon}
```

Defines an image with CSS class `icon`. The style is defined in `css/fancy.css`. The result:

Or define the style directly: `![](images/idea.svg){style="width:2cm"}`

4.2 Tables

Markdown tables are supported for organizing tabular data:

A	B	C
Table	with borders	inside

A	B	C
Table with borders outside		

A	B	C
Table without borders		

4.3 More Markdown Examples

Check 'm out: https://markdown-it.github.io/

5 Conclusion

This document demonstrates all Docco's capabilities.