Tidy

Keep your code tidy.

v0.1.0 August 7, 2023

https://github.com/Mc-Zen/tidy

Mc-Zen

Abstract

tidy is a package that generates documentation directly in Typst for your Typst modules. It parses docstring comments similar to javadoc and co. and can be used to easily build a reference section for each module.

Contents

I Introduction

Feed **tidy** your in-code documented source files and get beautiful documentation of all your functions printed out. Enjoy features like type annotations, links to other documented functions and arbitrary formatting within function and parameter descriptions. Let's get started.

First, we import **tidy**.

```
#import "@preview/tidy:0.1.0"
```

We now assume we have a Typst module called `my-module.typ`, containing a definition for a function named `something()`.

Example of some documented source code:

```
/// This function does something. It always returns true.
///
/// We can have *markdown* and
/// even $m^a t_h$ here. A list? No problem:
///   - Item one
///   - Item two
///
/// - param1 (string): This is param1.
/// - param2 (content, length): This is param2.
///             Yes, it really is.
/// - ..options (any): Optional options.
/// -> boolean, none
#let something(param1, param2: 3pt, ..options) = { return true }
```

You can document your functions similar to javadoc by prepending a block of `///` comments. Each line needs to start with three slashes `///` (whitespace is allowed at the beginning of the line). Parameters of the function can be documented by listing them as

```
/// - parameter-name (type): ...
```

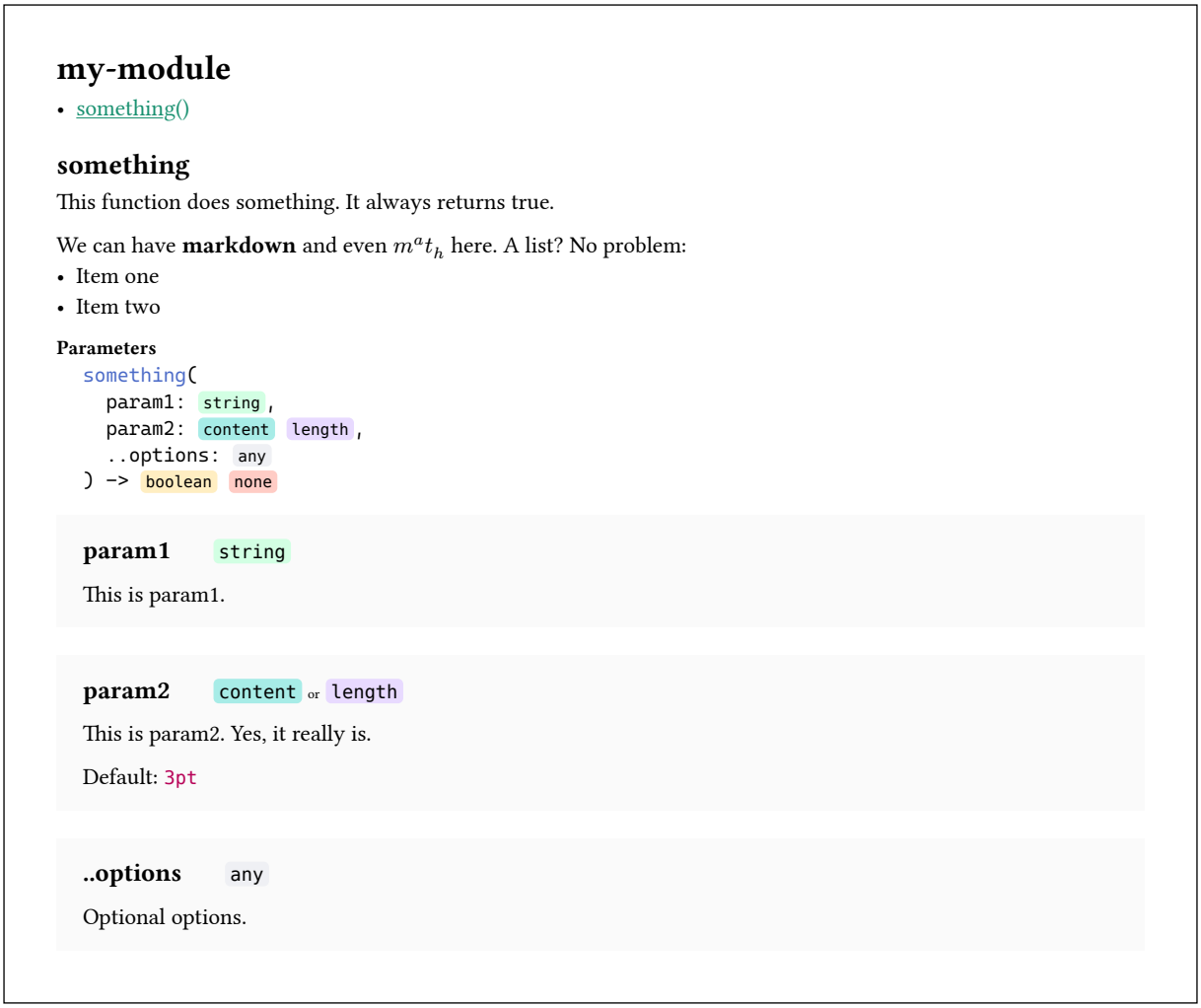
Following this exact form is important (see also the spaces marked in red) since this allows to distinguish the parameter list from ordinary markup lists in the function description or in parameter descriptions. For example, another space in front of the `-` could be added to markup lists if necessary.

The possible types for each parameter are given in parentheses and after a colon `:`, the parameter description follows. Indicating a type is mandatory (you may want to pick `any` in some cases). An optional return type can be annotated by ending with a line that contains `->` and the return type(s).

In front of the parameter list, a function description can be put. Both function and parameter descriptions may span multiple lines and can contain any Typst code (see Section II on how to use images, user-defined variables and functions in the docstring).

Calling `parse-module()` will read out the documentation of the given string. We can then invoke `show-module()` on the result.

```
#let my-module = tidy.parse-module(read("my-module.typ"), name: "my-module")
#tidy.show-module(my-module)
```



my-module

- something()

something

This function does something. It always returns true.

We can have **markdown** and even $m^a t_h$ here. A list? No problem:

- Item one
- Item two

Parameters

```
something(
  param1: string,
  param2: content length,
  ..options: any
) -> boolean none
```

param1 `string`

This is param1.

param2 `content` or `length`

This is param2. Yes, it really is.

Default: `3pt`

..options `any`

Optional options.

Cool, he?

By default, an outline for all functions is displayed at the top. This behaviour can be turned off with the parameter `show-outline` of `show-module()`.

There is another nice little feature: in the docstring, you can reference other functions with the extra syntax `@@other-function()`. This will automatically create a link that when clicked in the PDF will lead you to the documentation of that function.

Of course, everything happens instantaneously, so you can see the live result while writing the docs for your package. Keep your code documented!

II Accessing User-Defined Symbols

This package uses the Typst function `eval()` to process function and parameter descriptions in order to enable arbitrary Typst markup in them. Since `eval()` does not allow access to the filesystem and evaluates the content in a context where no user-defined variables or functions are available, it is not possible to directly call `#import`, `#image` or functions that you define in your code.

Nevertheless, definitions can be made accessible by passing them to `parse-module()` through the optional `scope` parameter in form of a dictionary:

```
#let make-square(width) = rect(width: width, height: width)

#tidy.parse-module(read("my-module.typ"), scope: (make-square: make-square))
```

This makes any symbol in specified in the `scope` dictionary available under the name of the key. A function declared in `my-module.typ` can now use this variable in the description:

```
/// This is a function
///
/// #make-square(20pt)
///
#let my-function() = {}
```

It is even possible to add **entire modules** to the scope which makes rendering examples using your module really easy. Let us say, `my-sine.typ` contains:

```
/// Draw a sine function with n periods into a rectangle of given size
///
/// *Example:*
///
/// #my-sine.draw-sine(1cm, 0.5cm, 2)
///
/// - height (length): Width of bounding rectangle.
/// - width (length): Height of bounding rectangle.
/// - periods (integer, float): Number of periods to draw.
///      Example with many periods: \ #my-sine.draw-sine(4cm, 0.3cm, 10)
/// -> content
#let draw-sine(width, height, periods) = box(width: width, height: height, {
  let prev-point = (0pt, height / 2)
  let res = 100
  for i in range(1, res) {
    let x = i / res * width
    let y = (1 - calc.sin(i / res * 2 * 3.1415926 * periods)) * height / 2
    place(line(start: prev-point, end: (x, y)))
    prev-point = (x, y)
  }
})
```

We can now parse the module and pass the module `my-sine` through the `scope` parameter:

```
#import "my-sine.typ" // don't import something specific from the module!

#let module = tidy.parse-module(
  read("my-sine.typ"),
  scope: (my-sine: my-sine)
)
```

draw-sine

Draw a sine function with n periods into a rectangle of given size

Example:

Parameters

```
draw-sine(
  width: length,
  height: length,
  periods: integer float
) -> content
```

width `length`

Height of bounding rectangle.

height `length`

Width of bounding rectangle.

periods `integer` or `float`

Number of periods to draw. Example with many periods:

III Customizing the Style

There are multiple ways to customize the output style. You can
- pick a different predefined style,
- apply show rules before printing the module documentation or
- create an entirely new style.

A different predefined style can be selected by passing a style to the `style` parameter:

```
#tidy.show-module(
  tidy.parse-module(read("my-module.typ")),
  style: tidy.styles.minimal
)
```

You can use show rules to customize the document style before calling `show-module()`. Setting any text and paragraph attributes works just out of the box. Furthermore, heading styles can be set to affect the appearance of the module name (relative heading level 1), function names (relative heading level 2) and the word **Parameters** (relative heading level 3), all relative to what is set with the parameter `first-heading-level` of `show-module()`.

Finally, if that is not enough, you can design a completely new style. Examples of styles can be found in the folder `src/styles/` in the GitHub Repository.

IV Function Documentation

Let us now “self-document” this package:

tidy

parse-module

Parse the docstrings of a typst module. This function returns a dictionary with the keys
- `name` : The module name as a string.
- `functions` : A list of function documentations as dictionaries.
- `label-prefix` : The prefix for internal labels and references.

The label prefix will automatically be the name of the module if not given explicity.

The function documentation dictionaries contain the keys
- `name` : The function name.
- `description` : The function's docstring description.
- `args` : A dictionary of info objects for each function argument.

These again are dictionaries with the keys
- `description` (optional): The description for the argument.
- `types` (optional): A list of accepted argument types.
- `default` (optional): Default value for this argument.

See `show-module()` for outputting the results of this function.

Parameters

```
parse-module(
  content: string,
  name: string,
  label-prefix: auto string,
  require-all-parameters: boolean,
  scope: dictionary
)
```

content `string`

Content of `.typ` file to analyze for docstrings.

name `string`

The name for the module.

Default: `""`

label-prefix `auto` or `string`

The label-prefix for internal function references. If `auto`, the label-prefix name will be the module name.

Default: `auto`

require-all-parameters `boolean`

Require that all parameters of a functions are documented and fail if some are not.

Default: `false`

scope `dictionary`

A dictionary of definitions that are then available in all function and parameter descriptions.

Default: `(:)`

show-module

Show given module in the given style. This displays all (documented) functions in the module.

Parameters

```
show-module(
  module-doc: dictionary,
  style: module dictionary,
  first-heading-level: integer,
  show-module-name: boolean,
  break-param-descriptions: boolean,
  omit-empty-param-descriptions: boolean,
  show-outline: function,
  sort-functions: auto none function
) -> content
```

module-doc `dictionary`

Module documentation information as returned by `parse-module()`.

style `module` or `dictionary`

The output style to use. This can be a module defining the functions `show-outline`, `show-type`, `show-function`, `show-parameter-list` and `show-parameter-block` or a dictionary with functions for the same keys.

Default: `styles.default`

first-heading-level `integer`

Level for the module heading. Function names are created as second-level headings and the “Parameters” heading is two levels below the first heading level.

Default: `2`

show-module-name `boolean`

Whether to output the name of the module at the top.

Default: `true`

break-param-descriptions `boolean`

Whether to allow breaking of parameter description blocks.

Default: `false`

omit-empty-param-descriptions `boolean`

Whether to omit description blocks for parameters with empty description.

Default: `true`

show-outline `function`

Whether to output an outline of all functions in the module at the beginning.

Default: `true`

sort-functions `auto` or `none` or `function`

Function to use to sort the function documentations. With `auto`, they are sorted alphabetatically by name and with `none` they are not sorted. Otherwise a function can be passed that each function documentation object is passed to and that should return some key to sort the functions by.

Default: `auto`