

# Tidy

Keep your code tidy.

v0.2.0

January 03, 2024

<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 .....	2
II Accessing user-defined symbols .....	5
III Preview examples .....	7
IV Customizing the style .....	8
V Docstring testing .....	10
VI Function documentation .....	11

# I INTRODUCTION

You can easily feed **tidy** your in-code documented source files and get beautiful documentation of all your functions and variables printed out. The main features are:

- Type annotations,
- Seamless cross references,
- Rendering code examples (see Section III), and
- Docstring testing (see Section V).

First, we import **tidy**.

```
1 #import "@preview/tidy:0.2.0"
```

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

repeater.typ	
1	/// Repeats content a specified number of times.
2	/// - body (content): The content to repeat.
3	/// - num (integer): Number of times to repeat the content.
4	/// - separator (content): Optional separator between repetitions
5	/// of the content.
6	/// -> content
7	#let repeat(body, num, separator: []) = ((body,)*num).join(separator)
8	
9	/// An awfully bad approximation of pi.
10	/// -> float
11	#let awful-pi = 3.14

A **function** is documented 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

```
1 /// - 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 `->` followed by 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).

**Variables** are documented just in the same way (lacking the option to specify parameters). A definition is recognized as a variable if the identifier (variable/function name) is not followed by an opening

parenthesis. The `->` syntax which also specifies the return type for functions can be used to define the type of a variable.

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

```
1 #let docs = tidy.parse-module(read("docs.typ"), name: "Repeater")
2 #tidy.show-module(docs)
```

This will produce the following output.

## Repeater

- `repeat()`

### Variables:

- `awful-pi()`

### repeat

Repeats content a specified number of times.

### Parameters

```
repeat(  
  body: content,  
  num: integer,  
  separator: content  
) -> content
```

**body** `content`

The content to repeat.

**num** `integer`

Number of times to repeat the content.

**separator** `content`

Optional separator between repetitions of the content.

Default: []

**awful-pi** `float`

An awfully bad approximation of pi.

Cool, he?

By default, an outline for all definitions 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 cross-reference other definitions with the extra syntax `@@repeat()` or `@@awful-pi`. This will automatically create a link that when clicked in the PDF will lead you to the documentation of that definition.

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 within those. 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 with **tidy** by passing them to `parse-module()` through the optional `scope` parameter in form of a dictionary:

```
1 #let make-square(width) = rect(width: width, height: width)
2 #tidy.parse-module(
3   read("my-module.typ"), scope: (make-square: make-square)
4 )
```

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:

```
1 /// This is a function
2 /// #make-square(20pt)
3 #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 the file `wiggly.typ` contains:

```
wiggly.typ
1 /// Draw a sine function with $n$ periods into a rectangle of given size.
2 ///
3 /// *Example:*
4 /// #example(`wiggly.draw-sine(1cm, 0.5cm, 2)`)
5 ///
6 /// - height (length): Width of bounding rectangle.
7 /// - width (length): Height of bounding rectangle.
8 /// - periods (integer, float): Number of periods to draw.
9 ///     Example with many periods:
10 ///     #example(`wiggly.draw-sine(4cm, 1.3cm, 10)`)
11 /// -> content
12 #let draw-sine(width, height, periods) = box(width: width, height: height, {
13   let resolution = 100
14   let frequency = 1 / resolution * 2 * calc.pi * periods
15   let prev-point = (0pt, height / 2)
16   for i in range(1, resolution) {
17     let x = i / resolution * width
18     let y = (1 - calc.sin(i * frequency)) * height / 2
19     place(line(start: prev-point, end: (x, y)))
20     prev-point = (x, y)
21   }
22 })
```

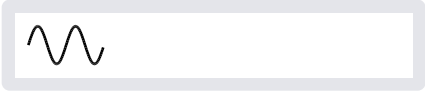
Note, that we use the predefined function `example()` here to show the code as well as the rendered output of some demo usage of our function. The `example()` function is treated more in-detail in Section III.

We can now parse the module and pass the module `wiggly` through the `scope` parameter:

```
1 #import "wiggly.typ" // don't import something specific from the module!
2
3 #let docs = tidy.parse-module(
4   read("wiggly.typ"),
5   name: "wiggly",
6   scope: (wiggly: wiggly)
7 )
```

In the output, the preview of the code examples is shown next to it.

**wiggly**  
**draw-sine**  
Draw a sine function with  $n$  periods into a rectangle of given size.  
**Example:**

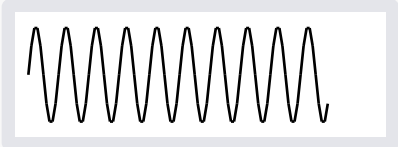
`wiggly.draw-sine(1cm, 0.5cm, 2)`

**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:

`wiggly.draw-sine(4cm, 1.3cm, 10)`

### III PREVIEW EXAMPLES

As we saw in the previous section, it is possible with **tidy** to add examples to a docstring and preview it along with its output. The function `example()` is available in every docstring and has some bells and whistles which are showcased with the following `example-demo.typ` module which contains a function for highlighting text with gradients (seems not very advisable due to the poor readability):

#### flashy

```
/// #example(`example-demo.flashy[We like our code flashy]`)
```

```
example-demo.flashy[We like our code flashy]
```

We like our code flashy

```
/// #example(`example-demo.flashy[Large previews will be scaled automatically to fit]`)
```

```
example-demo.flashy[Large previews will be  
scaled automatically to fit]
```

Large previews will be scaled automatically to fit

```
/// #example(`example-demo.flashy[Change code to preview ratio]`, ratio: 2)
```

```
example-demo.flashy[Change code to preview ratio]
```

Change code to preview ratio

```
/// #example(`example-demo.flashy(map: color.map.vlag)[Huge preview]`, scale-preview: 200%)
```

```
example-demo.flashy(map: color.map.vlag)[Huge  
preview]
```

Huge preview

```
/// #example(`flashy[Add to scope]`, scope: (flashy: example-demo.flashy, i: 2))
```

```
flashy[Add to scope #i ...]
```

Add to scope 2 ..

```
/// #example(`Markup *mode*`, mode: "markup")
```

```
Markup *mode*
```

Markup **mode**

```
/// #example(`e^(i phi) = -1`, mode: "math")
```

```
e^(i phi) = -1
```

$e^{i\varphi} = -1$

```
/// #example(`example-demo.flashy(map: color.map.crest)[Very extremely long examples might  
maybe require the need of vertical layouting]`, dir: ttb)
```

```
example-demo.flashy(map: color.map.crest)[Very extremely long examples might maybe require the need  
of vertical layouting]
```

Very extremely long examples might maybe require the need of vertical layouting

#### Parameters

```
flashy(  
  body: content,  
  map: array  
) -> content
```

body content

map array

Default: color.map.spectral

## IV 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:

```
1 #tidy.show-module(  
2     tidy.parse-module(read("my-module.typ")),  
3     style: tidy.styles.minimal  
4 )
```

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 or variable 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.a Customizing Colors (mainly for the `default` style)

The colors used by a style (especially the color in which types are shown) can be set through the option `colors` of `show-module()`. It expects a dictionary with colors as values. Possible keys are all type names as well as `signature-func-name` which sets the color of the function name as shown in a function signature.

The `default` theme defines a color scheme `colors-dark` along with the default `colors` which adjusts the plain colors for better readability on a dark background.

```
1 #tidy.show-module(  
2     docs,  
3     colors: tidy.styles.default.colors-dark  
4 )
```

With a dark background and light text, these colors produce much better contrast than the default colors:

```
space  
Produces space.  
  
Parameters  
space(amount: length)
```



## IV.b Predefined styles

Currently, the two predefined styles `tidy.styles.default` and `tidy-styles.minimal` are available.

- `tidy.styles.default`: Loosely imitates the online documentation of Typst functions.
- `tidy.styles.minimal`: A very light and space-efficient theme that is oriented around simplicity. With this theme, the example from above looks like the following:

### wiggly

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

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

#### Example:

```
wiggly.draw-sine(1cm, 0.5cm, 2)
```



#### Parameters:

`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:

```
wiggly.draw-sine(4cm, 1.3cm, 10)
```



## V Docstring Testing

Tidy supports small-scale docstring tests that are executed automatically and throw appropriate error messages when a test fails.

In every docstring, the function `test(., tests, scope: (:))` is available. An arbitrary number of tests can be passed in and the evaluation scope may be extended through the `scope` parameter. Any definition exposed to the docstring evaluation context through the `scope` parameter passed to `parse-module(.)` (see Section II) is also accessible in the tests. Let us create a module `num.typ` with the following content:

```
1  /// #test(  
2  ///   `num.my-square(2) == 4`,  
3  ///   `num.my-square(4) == 16`,  
4  /// )  
5  #let my-square(n) = n * n
```

Parsing and showing the module will run the docstring tests.

```
1  #import "num.typ"  
2  #let module = tidy.parse-module(  
3    read("num.typ"),  
4    name: "num",  
5    scope: (num: num)  
6  )  
7  #tidy.show-module(module) // tests are run here
```

As alternative to using `test()`, the following dedicated shorthand syntax can be used:

```
1  /// >>> my-square(2) == 4  
2  /// >>> my-square(4) == 16  
3  #let my-square(n) = n * n
```

When using the shorthand syntax, the error message even shows the line number of the failed test in the corresponding module.

A few test assertion functions are available to improve readability, simplicity and error messages. Currently, these are `eq(a, b)` for equality tests, `ne(a, b)` for inequality tests and `approx(a, b, eps: 1e-10)` for floating point comparisons. These assertion helper functions are always available within docstring tests (with both `test()` and `>>>` syntax).

## VI FUNCTION DOCUMENTATION

Let us now “self-document” this package:

### tidy

- [parse-module\(\)](#)
- [show-module\(\)](#)

#### 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 explicitly.

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,  
  enable-tests: boolean,  
  colors: auto dictionary  
) -> 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 alphabetically 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`

**enable-tests** `boolean`

Whether to run docstring tests.

Default: `true`

**colors** `auto` or `dictionary`

Give a dictionary for type and colors and other colors. If set to `auto`, the style will select its default color set.

Default: `auto`