Tidy

Keep your code tidy.

v0.3.0 May 13, 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

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 IV),
- help command generation (see Section VI), and
- Docstring testing (see Section VII).

First, we import **tidy**.

```
#import "@preview/tidy:0.3.0"
```

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

repeater.typ

```
/// Repeats content a specified number of times.
/// - body (content): The content to repeat.
/// - num (integer):  Number of times to repeat the content.
/// - separator (content): Optional separator between repetitions
///                        of the content.
/// -> content
#let repeat(body, num, separator: []) = ((body,)*num).join(separator)

/// An awfully bad approximation of pi.
/// -> float
#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

```
/// - 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 III 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.

```
#let docs = tidy.parse-module(read("docs.typ"), name: "Repeater")
#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 More options

Sometimes you want to document "private" functions and variables but omit them in the public documentation. In order to hide all definitions starting with an underscore, you may set `omit-private-definitions` to `true` in the call to `show-module()`. Similarly, "implementation parameters" to otherwise public functions occur once in a while. These are then used internally by the library. In order to conceal such parameters which may lead to confusion with dedicated documentation readers, you can name them with a leading underscore and set `omit-private-parameters` to `true` as well.

III 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:

```
#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 the file `wiggly.typ` contains:

wiggly.typ

```
/// Draw a sine function with $n$ periods into a rectangle of given size.
///
/// *Example:*
/// #example(`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:
///      #example(`draw-sine(4cm, 1.3cm, 10)`)
/// -> content
#let draw-sine(width, height, periods) = box(width: width, height: height, {
  let resolution = 100
  let frequency = 1 / resolution * 2 * calc.pi * periods
  let prev-point = (0pt, height / 2)
  for i in range(1, resolution) {
    let x = i / resolution * width
    let y = (1 - calc.sin(i * frequency)) * height / 2
    place(line(start: prev-point, end: (x, y)))
    prev-point = (x, y)
  }
})
```

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 IV.

We can now parse the module and pass the module `wiggly` through the `scope` parameter. Furthermore, we apply another trick: by specifying a `preamble`, we can add code to run before each example. Here we use this feature to import everything from the module `wiggly`. This way, we can directly write `draw-sine(...)` in the example (instead of `wiggly.draw-sine(...)`):

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

#let docs = tidy.parse-module(
  read("wiggly.typ"),
  name: "wiggly",
  scope: (wiggly: wiggly),
  preamble: "import wiggly: *;"
)
```

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

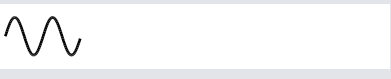
wiggly

draw-sine

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

Example:

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



Parameters

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

width `length`

Height of bounding rectangle.

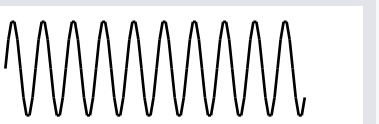
height `length`

Width of bounding rectangle.

periods `integer` or `float`

Number of periods to draw. Example with many periods:

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



IV 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`

V 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 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.

V.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.

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

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

space

Produces space.

Parameters

`space(amount: length)`

V.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:

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

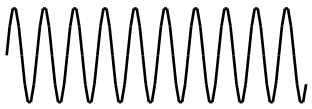
Parameters:

`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:

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

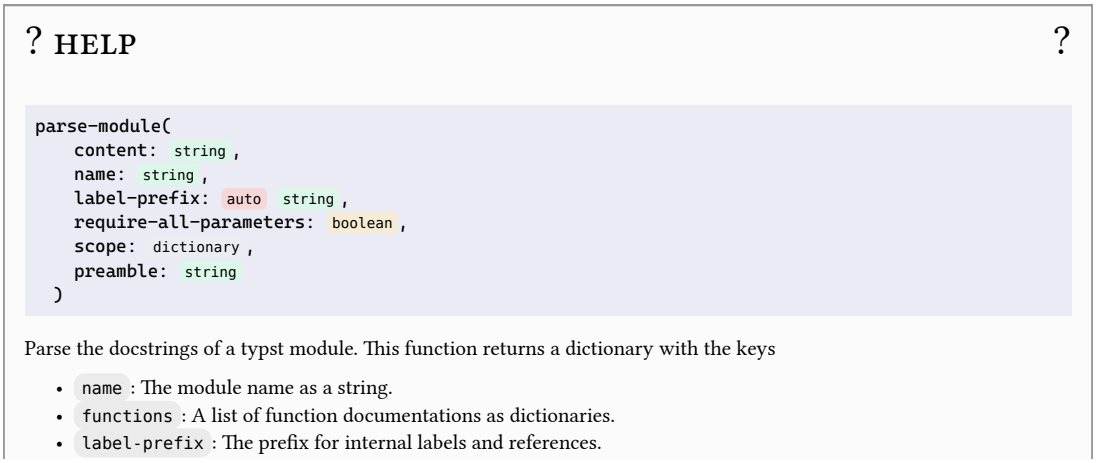


VI Help command

This feature is still experimental and may change a bit in its details. Output customization will be made available with the introduction of user-defined types into Typst. The *search* feature will then move into a nested function, i.e., `help.search()`.

With **tidy**, you can easily add a `help` command to your package. This allows the users of your package to call `#your-package.help("foo")` to get the docs for the specified definition printed right in their document. This makes reading up on options and discovering features in your package effortless. After the desired information has been gathered, it's no more than deleting a line of document source code to make the docs vanish into the hidden realms of repositories once again!

As a demonstration, calling `#tidy.help("parse-module")` produces the following (clipped) output into the document.



? HELP ?

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

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.

This feature supports:

- function and variable definitions,
- definitions defined in nested submodules, e.g., `#your-package.help("sub.bar.foo")`
- asking only for the parameter description of a function, e.g., `#your-package.help("foo(param1)")`
- lazy evaluation of docstring processing (even loading `tidy` is made lazy). *Don't pay for what you don't use!*
- search through the entire package documentation, e.g., `#your-package.help(search: "module")`

VI.a Setup

If you have already documented your code, adding such a help function will require only little further effort in implementation. In your root library file, add some code of the following kind:

```
#let help(..args) = {
  import "@preview/tidy:0.3.0"
  let namespace = (
    ".": read.with("/src/my-package.typ")
  )
  tidy.generate-help(namespace: namespace, package-name: "tidy")(..args)
}
```

First, we set up a `namespace` dictionary that reflects the way that definitions can be accessed by a user. Note that due to import statements that import *from* a file, this may not reflect the actual file structure of your repository. Take care to provide `read.with()` objects with the filename prepended instead of directly calling `read()`. This allows **tidy** to only lazily read the source files upon a help request from the end user.

As a more elaborate example, let us look at some library root file for a maths package called `heymath`.

heymath.typ

```
#import "vec.typ": vec-add, vec-subtract // import definitions into root
#import "matrix.typ"                     // submodule "matrix"

/// ...
#let pi-squared = 9.869604401089358618 83
```

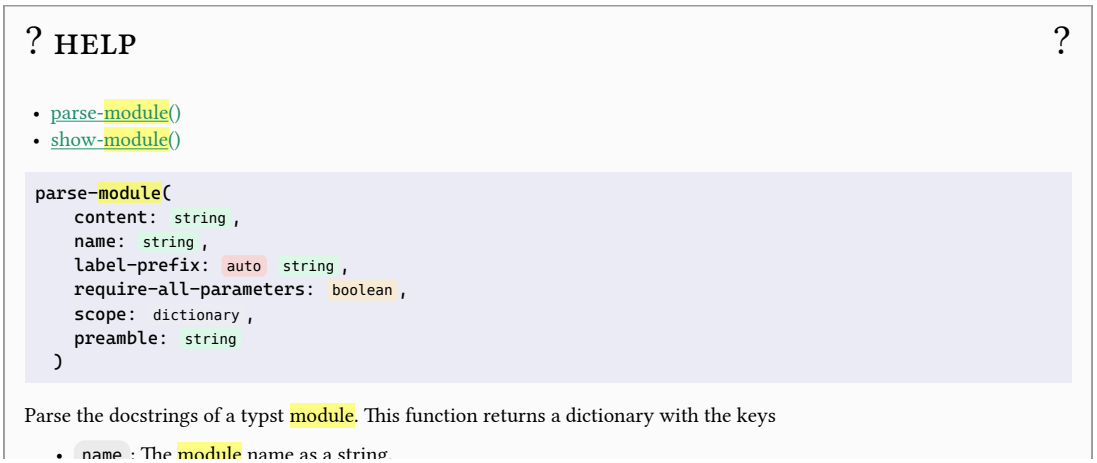
Our `namespace` dictionary could then look like this:

```
let namespace = (
  ".": (read.with("/heymath.typ"), read.with("/vec.typ"))
  "matrix": read.with("/matrix.typ")
  "matrix.solve": read.with("/solve.typ")
)
```

Since the symbols from `vec.typ` are imported directly into the library (and are accessible through `heymath.vec-add()` and `heymath.vec-subtract()`), we add this file to the root together with the main library file. Both files will be internally concatenated for docstring processing. The content of `matrix.typ`, however, can only be accessed through `heymath.matrix.` (by the user) and so we place `matrix.typ` at the key `matrix`. For nested submodules, write out the complete name "path" for the key. As an example, we have added `matrix.solve` – a module that would be imported within `matrix.typ` – to the code sample above. **It is advised not to change the signature of the help function manually in order to keep consistency between different packages using this features**.

VI.b Searching

It is also possible to search the package documentation via the search argument of the help function: `#tidy.help(search: "module")`. This feature is even more experimental.


? HELP ?

- parse-module()
- show-module()

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

Parse the docstrings of a typst module. This function returns a dictionary with the keys

- name : The module name as a string

VI.c Output customization (for end-users)

The default style for help output should work more or less for light and dark documents but is otherwise not very customizable. This is intended to be changed when user-defined types are available in Typst because these would provide the ideal interface for such customization. Until then, I do not deem it much sense to provide a temporary solution that need.

VI.d Notes about optimization (for package developers)

When set up in the form as shown above, the package `tidy` is only imported when a user calls `help` for the first time and not at all if the feature is not used *(don't pay for what you don't use)*. The files themselves are also only read when a definition from a specific submodule in the "namespace" is requested. In the case of *extremely* long code files, it *could* make sense to separate the documentation from the implementation by adding "documentation files" that only contain a *declaration* plus docstring for each definition – with the body left empty.

```
/// - inputs (array): The inputs for the algorithm.
/// - parameters (none, dictionary): Some parameters.
#let my-really-long-algorithm(inputs, parameters: none) = { }
```

The advantage is that the source code is not as crowded with (sometimes very long) docstrings and that docstring parsing may get faster. On the downside, there is an increased maintenance overhead due to the need of synchronizing the actual file and the documentation file (especially when the interface of a function changes).

VII 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 III) is also accessible in the tests. Let us create a module `num.typ` with the following content:

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

Parsing and showing the module will run the docstring tests.

```
#import "num.typ"
#let module = tidy.parse-module(
  read("num.typ"),
  name: "num",
  scope: (num: num)
)
#tidy.show-module(module) // tests are run here
```

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

```
/// >>> my-square(2) == 4
/// >>> my-square(4) == 16
#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).

Docstring tests can be disabled by passing `enable-tests: false` to `show-module()`.

VIII Function documentation

Let us now "self-document" this package:

tidy

- parse-module()
- show-module()
- generate-help()

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,
  preamble: string
)
```

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: `(:)`

preamble `string`

Code to prepend to all code snippets shown with `#example()`. This can for instance be used to import something from the scope.

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,
  omit-private-definitions: boolean,
  omit-private-parameters: boolean,
  show-outline: function,
  sort-functions: auto none function,
  enable-tests: boolean,
  enable-cross-references: boolean,
  colors: auto dictionary,
  local-names: 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`

omit-private-definitions `boolean`

Whether to omit functions and variables starting with an underscore.

Default: `false`

omit-private-parameters `boolean`

Whether to omit named function arguments starting with an underscore.

Default: `false`

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`

enable-cross-references `boolean`

Whether to enable links for cross-references.

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`

local-names `dictionary`

Language-specific names for strings used in the output. Currently, these are `parameters` and `default`. You can for example use: `local-names: (parameters: [Paramètres], default: [défault])`

Default: `(parameters: [Parameters], default: [Default])`

generate-help

Generates a `help` function for your package that allows the user to prints references directly into their document while typing. This allows them to easily check the usage and documentation of a function or variable.

Parameters

```
generate-help(
  namespace: dictionary,
  package-name: string,
  style: dictionary,
  onerror: function
)
```

namespace `dictionary`

This dictionary should reflect the "namespace" of the package in a flat dictionary and contain `read.with()` instances for the respective code files. Imagine importing everything from a package, `#import "mypack.typ": *`. How a symbol is accessible now determines how the dictionary should be built. We start with a root key, `(".": read.with("lib.typ"))`. If `lib.typ` imports symbols from other files *into* its scope, these files should be added to the root along with `lib.typ` by passing an array:

```
1  (
2    ".": (read.with("lib.typ"), read.with("more.typ")),
3    "testing": read.with("testing.typ")
4  )
```

Here, we already show another case: let `testing.typ` be imported in `lib.typ` but without `*`, so that the symbols are accessed via `testing.`. We therefore add these under a new key. Nested files should be added with multiple dots, e.g., `"testing.float."`.

By providing instances of `read()` with the filename prepended, you allow tidy to read the files that are not part of the tidy package but at the same time enable lazy evaluation of the files, i.e., a file is only opened when a definition from this file is requested through `help()`.

Default: `(".": () => "")`

package-name `string`

The name of the package. This is required to give helpful error messages when a symbol cannot be found.

Default: `""`

style `dictionary`

A tidy style that is used for showing parts of the documentation in the help box. It is recommended to leave this at the `help` style which is particularly designed for this purpose. Please post an issue if you have problems or suggestions regarding this style.

Default: `styles.help`

onerror `function`

What to do with errors. By default, an assertion is failed (the document panics).

Default: `msg => assert(false, message: msg)`