

Tidy Types

v0.0.1

<https://github.com/SillyFreak/typst-packages/tree/main/tidy-types>

Clemens Koza

ABSTRACT

Helpers for writing complex types in tidy documentation and rendering types like tidy outside of tidy-generated signatures.

CONTENTS

I Introduction	2
II Built-in Typst types	2
III Module reference	3

I INTRODUCTION

This package contains helpers for documenting the types of values with tidy, just as tidy itself shows them in function signatures. For example, this lets you write documentation such as this:

The result of `range(5).enumerate()` is a `((integer , integer) , ...)`.

To do so, it produces raw blocks with language "tidy-type", which can be then styled using a show rule as follows:

```
1 #import "@preview/tidy:0.2.0"
2 #import "@preview/tidy-types:0.1.0" as tt
3
4 // using the default style with default colors
5 #let style = tidy.styles.default
6 #show raw.where(lang: tt.lang): it => {
7   style.show-type(it.text, style-args: (colors: style.colors))
8 }
9
10 // using the minimal style
11 #let style = tidy.styles.minimal
12 #show raw.where(lang: tt.lang): it => style.show-type(it.text)
```

For example, the raw block ```tidy-type content``` would be displayed as `content` in the default style, or `content` in the minimal style. This can be more easily written using the basic function of this package, `tt.type():#tt.type("content")`. In practice, you will not need to use this function directly but instead use the utility functions and variables built on top.

II BUILT-IN TYPST TYPES

There are constants for all the built-in types that Typst provides. Note how two of them are prefixed with "t-" as their names are keywords – `none` and `auto` – and another because its name is taken by a tidy types function – `type`:

t-none	none	bool	boolean	int	integer
float	float	str	string	bytes	bytes
array	array	dictionary	dictionary	t-type	type
function	function	t-auto	auto	datetime	datetime
duration	duration	regex	regex	version	version
content	content	symbol	symbol	length	length
ratio	ratio	relative	relative length	fraction	fraction
angle	angle	color	color	stroke	stroke
alignment	alignment	location	location	styles	styles
label	label	selector	selector	module	module
plugin	plugin	arguments	arguments		

III MODULE REFERENCE

III.a tidy-types

- [type\(\)](#)
- [arr\(\)](#)
- [dict\(\)](#)
- [tuple\(\)](#)
- [object\(\)](#)
- [func\(\)](#)
- [either\(\)](#)
- [optional\(\)](#)

```
type(text: string) -> content
```

Wraps the given string, a type name, into a raw element with the language "tidy-type". By itself, that doesn't do anything, but it allows styling that text using a [show](#) rule; see [the introduction](#).

1	<code>tt.type("foo")</code>	<code>foo</code>
---	-----------------------------	------------------

Parameters:

`text (string)` – the type name

```
arr(element: content) -> content
```

A function for rendering an array type including element type information:

1	<code>tt.arr(tt.int)</code>	<code>(integer , ...)</code>
---	-----------------------------	--------------------------------

This representation uses the array spread syntax to convey that there may be any number of `integer` elements in the array.

The name of this function is `arr` because `tt.array (array)` exists already.

Parameters:

`element (content)` – the element type of the array

```
dict(value: content) -> content
```

A function for rendering a dictionary type including element type information:

1	<code>tt.dict(tt.int)</code>	<code>(string : integer , ...)</code>
---	------------------------------	---

This representation uses the implicit `string` key type to convey that there may be any number of mappings in the dictionary.

The name of this function is `dict` because `tt.dictionary (dictionary)` exists already.

Parameters:

`value (content)` – the value type of the dictionary

```
tuple(..elements: content) -> content
```

A function for rendering an array type containing exactly the given elements:

1 <code>tt.tuple(tt.str, tt.int)</code>	<code>(string, integer)</code>
---	--------------------------------

Parameters:

`..elements (content)` – the tuple element types given as positional parameters

```
object(..pairs: content) -> content
```

A function for rendering a dictionary type containing exactly the given pairs:

1 <code>tt.object(a: tt.str, b: tt.int)</code>	<code>(a: string, b: integer)</code>
--	--------------------------------------

Parameters:

`..pairs (content)` – the object attribute name/type pairs given as named parameters

```
func(..args: content) -> content
```

A function for rendering a function type taking the given parameters and having the given return type:

1 <code>tt.func(2 tt.str, opt: tt.bool, 3 tt.int)</code>	<code>(string, opt: boolean) -> integer</code>
---	---

Note that the relative order of positional and named parameters is not preserved; all named parameters come after all positional parameters. It makes sense to, as a convention, put the result type after any named parameters.

The name of this function is `func` because `tt.function (function)` exists already.

Parameters:

`..args (content)` – the function parameter types and return type (last positional argument) of the function

```
either(..options: content) -> content
```

A function for rendering a choice between the given types:

```
1 tt.either(tt.str, tt.int)
```

```
string | integer
```

Parameters:

`..options (content)` – the possible types given as positional parameters

```
optional(type: content) -> content
```

A function for rendering a parameter/element that may be omitted

```
1 tt.optional(tt.str)
```

```
2 parbreak()
```

```
3 tt.func(tt.optional(tt.str), tt.str)
```

```
string ?
```

```
( string ? ) -> string
```

Parameters:

`type (content)` – the tuple element types given as positional parameters