



## Tools For Typst

v0.2.0

2023-08-12

MIT

A utility package for typst package authors

Jonas NEUGEBAUER

<https://github.com/jneug/typst-tools4typst>

**Tools for Typst** (t4t in short) is a utility package for Typst package and template authors. It provides solutions to some recurring tasks in package development.

The package can be imported or any useful parts of it copied into a project. It is perfectly fine to treat t4t as a snippet collection and to pick and choose only some useful functions. For this reason, most functions are implemented without further dependencies.

Hopefully, this collection will grow over time with **Typst** to provide solutions for common problems.

## Table of contents

N.1. Test functions .....	2
N.1.1. Command reference .....	2
N.2. Default values .....	9
N.2.1. Command reference .....	10
N.3. Assertions .....	14
N.3.1. Command reference .....	14
N.4. Element helpers .....	17
N.4.1. Command reference .....	17
N.5. Math functions .....	19
N.5.1. Command reference .....	19
N.6. Alias functions .....	21

## I. Index

## N.1. Test functions

```
#import "@preview/t4t:0.2.0": is
```

These functions provide shortcuts to common tests like `#is.eq()`. Some of these are not shorter as writing pure typst code (e.g. `a == b`), but can easily be used in `.any()` or `.find()` calls:

```
1 // check all values for none
2 if some-array.any(is-none) {
3   ...
4 }
5
6 // find first not none value
7 let x = (none, none, 5, none).find(is.not-none)
8
9 // find position of a value
10 let pos-bar = args.pos().position(is.eq.with("|"))
```

There are two exceptions: `#is-none()` and `#is-auto()`. Since keywords can't be used as function names, the `is` module can't define a function to do `is.none()`. Therefore the functions `#is-none()` and `#is-auto()` are provided in the base module of `t4t`:

```
#import "@preview/t4t:0.2.0": is-none, is-auto
```

The `is` submodule still has these tests, but under different names (`#is.n()` and `#is.non()` for `none` and `#is.a()` and `#is.aut()` for `auto`).

### N.1.1. Command reference

<code>#a()</code>	<code>#elem()</code>	<code>#none-of-type()</code>
<code>#all-of-type()</code>	<code>#empty()</code>	<code>#not-a()</code>
<code>#any()</code>	<code>#eq()</code>	<code>#not-any()</code>
<code>#any-type()</code>	<code>#has()</code>	<code>#not-auto()</code>
<code>#arr()</code>	<code>#label()</code>	<code>#not-empty()</code>
<code>#aut()</code>	<code>#loc()</code>	<code>#not-n()</code>
<code>#bool()</code>	<code>#n()</code>	<code>#not-none()</code>
<code>#color()</code>	<code>#neg()</code>	<code>#one-not-none()</code>
<code>#content()</code>	<code>#neq()</code>	<code>#same-type()</code>
<code>#dict()</code>	<code>#non()</code>	<code>#sequence()</code>

#### `#is.neg(test)`

Creates a new test function, that is true, when `test` is false.

Can be used to create negations of tests like:

```
#let not-row = is.neg(is.raw)
```

Argument

`test`

function | boolean

### 0.1.1 Test functions

test to negate.

#### **#is.eq(compare, value)**

Tests if values compare and value are equal.

Argument

compare

any

first value

Argument

value

any

second value

#### **#is.neq(compare, value)**

Tests if values compare and value are not equal.

Argument

compare

any

first value

Argument

value

any

second value

#### **#is.n(..values)**

Tests if any one of values is equal to none.

Argument

..values

any

values to test

#### **#is.non()**

Alias for n().

#### **#is.not-none(..values)**

### 0.1.1 Test functions

Tests if none of values is equal to none.

Argument

`..values`

any

values to test

`#is.not-n()`

Alias for `not-n()`

`#is.one-not-none(..values)`

Tests, if at least one value in `values` is not equal to none.

Useful for checking mutiple optoinal arguments for a valid value:

```
#if is.one-not-none(..args.pos()) [  
  #args.pos().find(is.not-none)  
]
```

Argument

`..values`

any

values to test

`#is.a(..values)`

Tests if any one of values is equal to auto.

Argument

`..values`

any

values to test

`#is.aut()`

Alias for `a()`

`#is.not-auto(..values)`

Tests if none of values is equal to auto.

Argument

`..values`

any

values to test

### 0.1.1 Test functions

**#is.not-a()**

Alias for not-auto()

**#is.empty(value)**

Tests, if value is *empty*.

A value is considered *empty* if it is an empty array, dictionary or string, or the value none.

Argument

value

any

value to test

**#is.not-empty(value)**

Tests, if value is not *empty*.

See empty() for an explanation what *empty* means.

Argument

value

any

value to test

**#is.any(..compare, value)**

Tests, if value is not *empty*.

See empty() for an explanation what *empty* means.

Argument

value

any

value to test

**#is.not-any(..compare, value)**

Tests if value is not equals to any one of the other passed in values.

Argument

..compare

any

values to compare to

Argument

value

any

value to test

### 0.1.1 Test functions

#### #is.has(..keys, value)

Tests if value contains all the passed keys.

Either as keys in a dictionary or elements in an array. If value is neither of those types, false is returned.

Argument

..keys

any

keys or values to look for

Argument

value

any

value to test

#### #is.type(t, value)

Tests if value is of type t.

Argument

t

string

name of the type

Argument

value

any

value to test

#### #is.dict(value)

Tests if value is of type dictionary.

Argument

value

any

value to test

#### #is.arr(value)

Tests if value is of type array.

Argument

value

any

value to test

### 0.1.1 Test functions

#### #is.content(value)

Tests if value is of type content.

Argument	
value	any
value to test	

#### #is.label(value)

Tests if value is of type label.

Argument	
value	any
value to test	

#### #is.color(value)

Tests if value is of type color.

Argument	
value	any
value to test	

#### #is.stroke(value)

Tests if value is of type stroke.

Argument	
value	any
value to test	

#### #is.loc(value)

Tests if value is of type location.

Argument	
value	any
value to test	

#### #is.bool(value)

Tests if value is of type boolean.

### 0.1.1 Test functions

Argument

value

any

value to test

**#is.any-type(..types, value)**

Tests if types value is any one of types.

Argument

..types

string

type names to check against

Argument

value

any

value to test

**#is.same-type(..values)**

Tests if all passed in values have the same type.

Argument

..values

any

values to test

**#is.all-of-type(t, ..values)**

Tests if all of the passed in values have the type t.

Argument

t

string

type to test against

Argument

..values

any

values to test

**#is.none-of-type(t, ..values)**

Tests if none of the passed in values has the type t.

Argument

t

string



type to test against

Argument

`..values`

any

values to test

`#is.elem(func, value)`

Tests if `value` is a content element with `value.func() == func`.

If `func` is a string, `value` will be compared to `repr(value.func())`, instead. Both of these effectively do the same:

```
#is.elem(raw, some_content)
#is.elem("raw", some_content)
```

Argument

`func`

function

element function

Argument

`value`

any

value to test

`#is.sequence(value)`

Tests if `value` is a sequence of content.

## N.2. Default values

```
#import "@preview/t4t:0.2.0": def
```

These functions perform a test to decide, if a given `value` is *invalid*. If the test *passes*, the default is returned, the value otherwise.

Almost all functions support an optional `do` argument, to be set to a function of one argument, that will be applied to the value, if the test fails. For example:

```

1 // Sets date to a datetime from an optional
2 // string argument in the format "YYYY-MM-DD"
3 let date = def.if-none(
4   datetime.today(), // default
5   passed_date, // passed in argument
6   do: (d) => { // post-processor
7     d = d.split("-")
8     datetime(year=d[0], month=d[1], day=d[2])
9   }
10 )

```

## N.2.1. Command reference

<code>#as-arr()</code>	<code>#if-auto()</code>	<code>#if-none()</code>
<code>#if-any()</code>	<code>#if-empty()</code>	<code>#if-not-any()</code>
<code>#if-arg()</code>	<code>#if-false()</code>	<code>#if-true()</code>

**#def.if-true(test, default, do: none, value)**

Returns default if test is true, value otherwise.

If test is false and do is set to a function, value is passed to do, before being returned.

Argument

<b>test</b>	boolean
a test result	

Argument

<b>default</b>	any
default value to return	

Argument

<b>do: none</b>	function
postprocessor for value	

Argument

<b>value</b>	any
the value to test	

**#def.if-false(test, default, do: none, value)**

Returns default if test is false, value otherwise.

If test is true and do is set to a function, value is passed to do, before being returned.

Argument

<b>test</b>	boolean
a test result	

### 0.2.1 Default values

Argument	
default	any
default value to return	

Argument	
do: none	function
postprocessor for value	

Argument	
value	any
the value to test	

**#def.if-none(default, do: none, value)**

Returns default if value is none, value otherwise.

If value is not none and do is set to a function, value is passed to do, before being returned.

Argument	
default	any
default value to return	

Argument	
do: none	function
postprocessor for value	

Argument	
value	any
the value to test	

**#def.if-auto(default, do: none, value)**

Returns default if value is auto, value otherwise.

If value is not auto and do is set to a function, value is passed to do, before being returned.

Argument	
default	any
default value to return	

Argument	
do: none	function
postprocessor for value	

Argument	
----------	--

## 0.2.1 Default values

value

any

the value to test

```
#def.if-any(..compare, default, do: none, value)
```

Returns default if value is equal to any value in compare, value otherwise.

```
#def.if-any(  
  none, auto,    // ..compare  
  1pt,          // default  
  thickness     // value  
)
```

If value is in compare and do is set to a function, value is passed to do, before being returned.

Argument

..compare

any

list of values to compare value to

Argument

default

any

default value to return

Argument

do: none

function

postprocessor for value

Argument

value

any

value to test

```
#def.if-not-any(..compare, default, do: none, value)
```

Returns default if value is not equal to any value in compare, value otherwise.

```
#def.if-not-any(  
  left, right, top, bottom, // ..compare  
  left,                    // default  
  position                 // value  
)
```

If value is in compare and do is set to a function, value is passed to do, before being returned.

Argument

..compare

any

list of values to compare value to

Argument

## 0.2.1 Default values

default	any
default value to return	

Argument	
do: none	function
postprocessor for value	

Argument	
value	any
value to test	

**#def.if-empty**(default, do: none, value)

Returns default if value is empty, value otherwise.

If value is not empty and do is set to a function, value is passed to do, before being returned.

Depends on `is.empty()`. See there for an explanation of *empty*.

Argument	
default	any
default value to return	

Argument	
do: none	function
postprocessor for value	

Argument	
value	any
value to test	

**#def.if-arg**(default, do: none, args, key)

Returns default if key is not an existing key in `args.named()`, `args.named().at(key)` otherwise.

If value is not in `args` and `do` is set to a function, the value is passed to `do`, before being returned.

Argument	
default	any
default value to return	

Argument	
----------	--

`do: none``function`

postprocessor for value

—Argument—

`args``any`

arguments to test

—Argument—

`key``any`

key to look for

**#def.as-arr(..values)**

Always returns an array containing all values.

Any arrays in values will be flattened into the result. This is useful for arguments, that can have one element or an array of elements:

```
#def.as-arr(author).join(", ")
```

## N.3. Assertions

```
#import "@preview/t4t:0.2.0": assert
```

This submodule overloads the default assert function and provides more asserts to quickly check if given values are valid. All functions use assert in the background.

Since a module in Typst is not callable, the assert function is now available as `#assert.that()`. `#assert.eq()` and `#assert.ne()` work as expected.

All assert functions take an optional argument `message` to set the error message for a failed assertion.

### N.3.1. Command reference

`#all-of-type()``#ne()``#not-any-type()``#any()``#neq()``#not-none()``#any-type()``#new()``#that()``#eq()``#not-any()``#that-not()`**#assert.that()**

Asserts that test is `true`. See assert

```
#assert.that-not(test, message: "")
```

Asserts that test is **false**.

Argument	
test	boolean
Assertion to test.	

Argument	
message: ""	str
A message to show if the assert fails.	

```
#assert.eq()
```

Asserts that two values are equal.

```
#assert.ne()
```

Asserts that two values are not equal.

```
#assert.neq()
```

Alias for ne()

```
#assert.not-none()
```

Asserts that a value is not **none**

```
#assert.any(..values, value, message: "")
```

Assert that value is any one of values.

Argument	
..values	any
A set of values to compare value to.	

Argument	
value	any
Value to compare.	

Argument	
message: ""	str
A message to show if the assert fails.	

### 0.3.1 Assertions

**#assert.not-any(..values, value, message: "")**

Assert that value is not any one of values.

assert.not-any(none, auto, 3)

—Argument—

**..values**

any

A set of values to compare value to.

—Argument—

**value**

any

Value to compare.

—Argument—

**message: ""**

str

A message to show if the assert fails.

**#assert.any-type(..types, value, message: "")**

Assert that values type is any one of types.

—Argument—

**..types**

string

A set of types to compare the type of value to.

—Argument—

**value**

any

Value to compare.

—Argument—

**message: ""**

str

A message to show if the assert fails.

**#assert.not-any-type(..types, value, message: "")**

Assert that values type is not any one of types.

—Argument—

**..types**

string

A set of types to compare the type of value to.

—Argument—

**value**

any

Value to compare.



Argument  
`message: ""` str  
 A message to show if the assert fails.

`#assert.all-of-type(t, ..values, message: "")`  
 Assert that the types of all values are equal to t.

Argument  
`t` string  
 The type to test against.

Argument  
`..values` any  
 Values to test.

Argument  
`message: ""` str  
 A message to show if the assert fails.

`#assert.new(test)`  
 Creates a new assertion from test.  
 The new assertion will take a value and pass it to test. test should return a boolean.

Argument  
`test` function  
 A test function: (string) => boolean

## N.4. Element helpers

```
#import "@preview/t4t:0.2.0": get
```

This submodule is a collection of functions, that mostly deal with content elements and *get* some information from them. Though some handle other types like dictionaries.

### N.4.1. Command reference

<code>#args()</code>	<code>#dict-merge()</code>	<code>#stroke-thickness()</code>
<code>#dict()</code>	<code>#stroke-paint()</code>	<code>#text()</code>

## 0.4.1 Element helpers

### `#get.dict(..dicts)`

Create a new dictionary from the passed values.

All named arguments are stored in the new dictionary as is. All positional arguments are grouped in key/value-pairs and inserted into the dictionary:

```
#get.dict("a", 1, "b", 2, "c", d:4, e:5)
// gives {a:1, b:2, c:none, d:4, e:5}
```

### `#get.dict-merge(..dicts)`

Recursively merges the passed in dictionaries.

```
#get.dict-merge(
  {a: 1},
  {a: {one: 1, two:2}},
  {a: {two: 4, three:3}}
)
// gives {a:{one:1, two:4, three:3}}
```

Based on work by @johannes-wolf for johannes-wolf/typst-canvas.

### `#get.args(args, prefix: "")`

Creates a function to extract values from an argument sink args.

The resulting function takes any number of positional and named arguments and creates a dictionary with values from `args.named()`. Positional arguments to the function are present in the result, if they are present in `args.named()`. Named arguments are always present, either with their value from `args.named()` or with the provided value.

A prefix can be specified, to extract only specific arguments. The resulting dictionary will have all keys with the prefix removed, though.

```
#let my-func( ..options, title ) = block(
  ..get.args(options)(
    "spacing", "above", "below",
    width:100%
  )
)[
  #text(..get.args(options, prefix:"text-")(
    fill:black, size:0.8em
  ), title)
]

#my-func(
  width: 50%,
  text-fill: red, text-size: 1.2em
)[#lorem(5)]
```

### `#get.text(element, sep: "")`

### 0.4.1 Element helpers

Recursively extracts the text content of a content element.

If present, all child elements are converted to text and joined with sep.

`#get.stroke-paint(stroke, default: rgb("#000000"))`

Returns the color of stroke. If no thickness information is available, default is used. **Deprecated since Typst 0.7.0:** use `stroke.thickness` instead.

Based on work by @PgBiel for PgBiel/typst-tablex.

`#get.stroke-thickness(stroke, default: 1pt)`

Returns the thickness of stroke. If no thickness information is available, default is used.

**Deprecated since Typst 0.7.0:** use `stroke.thickness` instead.

## N.5. Math functions

```
#import "@preview/t4t:0.2.0": math
```

Some functions to complement the native `calc` module.

### N.5.1. Command reference

`#clamp()`

`#lerp()`

`#map()`

`#math.minmax(a, b)`

Returns an array with the minimum of `a` and `b` as the first element and the maximum as the second:

```
#let (min, max) = math.minmax(a, b)
```

Argument

`a`

any

First value.

Argument

`b`

any

Second value.

`#math.clamp(min, max, value)`

### 0.5.1 Math functions

Clamps a value between min and max.

In contrast to `clamp()` this function works for other values than numbers, as long as they are comparable.

```
text-size = math.clamp(0.8em, 1.2em, text-size)
```

Argument

min

any

Maximum for value.

Argument

value

any

The value to clamp.

**#math.lerp(min, max, t)**

Calculates the linear interpolation of t between min and max.

t should be a value between 0 and 1, but the interpolation works with other values, too. To constrain the result into the given interval, use `clamp()`:

```
#let width = math.lerp(0%, 100%, x)
#let width = math.lerp(0%, 100%, math.clamp(0, 1, x))
```

Argument

min

any

Maximum for value.

Argument

t

float

Interpolation parameter .

```
#math.map(
  min,
  max,
  range-min,
  range-max,
  value
)
```

Maps a value from the interval [min, max] into the interval [range-min, range-max]:

```
#let text-weight = int(math.map(8pt, 16pt, 400, 800, text-size))
```

### 0.5.1 Math functions

The types of `min`, `max` and `value` have to be the same. The types of `range-min` and `range-max` also.

Argument  
`min` any  
Maximum of the initial interval.

Argument  
`range-min` any  
Maximum of the target interval.

Argument  
`value` any  
The value to map.

## N.6. Alias functions

```
#import "@preview/t4t:0.2.0": alias
```

Some of the native Typst function as aliases, to prevent collisions with some common argument namens.

For example using `numbering` as an argument is not possible, if the value is supposed to be passed to the `#numbering()` function. To still allow argument names, that are in line with the common Typst names (like `type`, `align` ...), these alias functions can be used:

```
1 #let exercise( no, numbering: "1" ) = [  
2   Exercise #alias.numbering(numbering, no)  
3 ]
```

The following functions have aliases right now:

- `numbering`
- `align`
- `type`
- `label`
- `text`
- `raw`
- `table`
- `list`
- `enum`
- `terms`
- `grid`
- `stack`
- `columns`

## Part I.

## Index

### A

`#a` ..... 2, 4  
`#all-of-type` ..... 8, 17  
`#any` ..... 5, 15  
`#any-type` ..... 8, 16  
`#args` ..... 18  
`#arr` ..... 6  
`#as-arr` ..... 14  
`#aut` ..... 2, 4

### B

`#bool` ..... 7

### C

`#clamp` ..... 19  
`#color` ..... 7  
`#content` ..... 7

### D

`#dict` ..... 6, 18  
`#dict-merge` ..... 18

### E

`#elem` ..... 9  
`#empty` ..... 5  
`#eq` ..... 2, 3, 15

### H

`#has` ..... 6

### I

`#if-any` ..... 12  
`#if-arg` ..... 13  
`#if-auto` ..... 11  
`#if-empty` ..... 13  
`#if-false` ..... 10  
`#if-none` ..... 11  
`#if-not-any` ..... 12  
`#if-true` ..... 10  
`#is-auto` ..... 2  
`#is-none` ..... 2

### L

`#label` ..... 7  
`#lerp` ..... 20  
`#loc` ..... 7

### M

`#map` ..... 20  
`#minmax` ..... 19

### N

`#n` ..... 2, 3  
`#ne` ..... 14, 15  
`#neg` ..... 2  
`#neq` ..... 3, 15  
`#new` ..... 17  
`#non` ..... 2, 3  
`#none-of-type` ..... 8  
`#not-a` ..... 5  
`#not-any` ..... 5, 16  
`#not-any-type` ..... 16  
`#not-auto` ..... 4  
`#not-empty` ..... 5  
`#not-n` ..... 4  
`#not-none` ..... 3, 15  
`#numbering` ..... 21

### O

`#one-not-none` ..... 4

### S

`#same-type` ..... 8  
`#sequence` ..... 9  
`#stroke` ..... 7  
`#stroke-paint` ..... 19  
`#stroke-thickness` ..... 19

### T

`#text` ..... 18  
`#that` ..... 14  
`#that-not` ..... 15  
`#type` ..... 6