



Tools For Typst

v0.3.1

2023-09-14

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

I. Usage

I.1. Load from package repository (Typst 0.6.0 and later)	2
I.2. Manual	2

II. Module reference

II.1. Test functions	3
II.1.1. Command reference	3
II.2. Default values	9
II.2.1. Command reference	9
II.3. Assertions	13
II.3.1. Command reference	13
II.4. Element helpers	19
II.4.1. Command reference	19
II.5. Math functions	24
II.5.1. Command reference	24
II.6. Alias functions	26

III. Index

Part I.

Usage

I.1. Load from package repository (Typst 0.6.0 and later)

For Typst 0.6.0 and later, the package can be imported from the *preview* repository:

```
#import "@preview/t4t:0.3.1": automaton
```

Alternatively, the package can be downloaded and saved into the system dependent local package repository.

Either download the current release from GitHub¹ and unpack the archive into your system dependent local repository folder² or clone it directly:

```
git clone https://github.com/jneug/typst-tools4typst.git t4t/0.3.1
```

In either case, make sure the files are placed in a subfolder with the correct version number: `t4t/0.3.1`

After installing the package, just import it inside your `typ` file:

```
#import "@local/t4t:0.3.1": automaton
```

I.2. Manual

The manual is created using **TIDY**³ with the **MANTYS**⁴ template.

TIDY will be loaded from the package repository while **MANTYS** needs to be installed manually into the local package repository. Refer to the **MANTYS** manual for further information.

The manual doubles as a test suite by adding simple tests to the docstring of each function.

¹<https://github.com/jneug/typst-tools4typst>

²<https://github.com/typst/packages#local-packages>

³<https://github.com/Mc-Zen/tidy>

⁴<https://github.com/jneug/typst-mantys>

Part II.

Module reference

II.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 than 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 like `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`).

II.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) → function`

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

Can be used to create negations of tests like:

2.1.1 Test functions

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

Argument

test

function | boolean

Test to negate.

#is.eq(compare, value) → boolean

Tests if values compare and value are equal.

Argument

compare

any

first value

Argument

value

any

second value

#is.neq(compare, value) → boolean

Tests if compare and value are not equal.

Argument

compare

any

First value.

Argument

value

any

Second value.

#is.n(..values) → boolean

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) → boolean

Tests if none of values is equal to **none**.

Argument

..values

any

Values to test.

#is.not-n()

Alias for #not-none()

2.1.1 Test functions

#is.one-not-none(..values) → boolean

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

Useful for checking mutiple optional 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) → boolean

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) → boolean

Tests if none of `values` is equal to `auto`.

Argument

`..values`

any

Values to test.

#is.not-a()

Alias for `#not-auto()`

#is.empty(value) → boolean

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) → boolean

Tests, if `value` is not *empty*.

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

Argument

`value`

any

value to test

#is.any(..compare, value) → boolean

2.1.1 Test functions

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)` → **boolean**

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	

`#is.has(..keys, value)` → **boolean**

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

2.1.1 Test functions

value to test

#is.arr(value)

Tests if value is of type array.

Argument

value

any

value to test

#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.

Argument

value

any

2.1.1 Test functions

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:

2.1.1 Test functions

```
#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.

II.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 )
```

II.2.1. Command reference

`#as-arr()`

`#if-auto()`

`#if-none()`

`#if-any()`

`#if-empty()`

`#if-not-any()`

`#if-arg()`

`#if-false()`

`#if-true()`

#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

test

boolean

2.2.1 Default values

A test result.

Argument

default

any

A default value.

Argument

do: none

function

Post-processor for value: (any) => any

Argument

value

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

test

boolean

A test result.

Argument

default

any

A default value.

Argument

do: none

function

Post-processor for value: (any) => any

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

A default value.

Argument

do: none

function

2.2.1 Default values

Post-processor for value: (any) => any

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

A default value.

Argument

do: none

function

Post-processor for value: (any) => any

Argument

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

A default value.

Argument

do: none

function

Post-processor for value: (any) => any

Argument

value

any

2.2.1 Default values

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

default

any

A default value.

Argument

do: none

function

Post-processor for value: (any) => any

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

A default value.

Argument

do: none

function

Post-processor for value: (any) => any

Argument

value

any

value to test

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

2.2.1 Default values

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
A default value.

Argument
`do: none` function
Post-processor for value: (any) => any

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` are unpacked into the resulting array.

This is useful for arguments, that can have one element or an array of elements:

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

II.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.

II.3.1. Command reference

2.3.1 Assertions

<code>#all-of-type()</code>	<code>#ne()</code>	<code>#not-any()</code>
<code>#any()</code>	<code>#neq()</code>	<code>#not-any-type()</code>
<code>#any-type()</code>	<code>#new()</code>	<code>#not-empty()</code>
<code>#eq()</code>	<code>#no-named()</code>	<code>#not-none()</code>
<code>#has-named()</code>	<code>#no-pos()</code>	<code>#that()</code>
<code>#has-pos()</code>	<code>#none-of-type()</code>	<code>#that-not()</code>

`#assert.that(test, message: "Test returned false, should be true.")`

Asserts that test is **true**. See `assert`.

Argument
test boolean
Assertion to test.

Argument
message: "Test returned false, should be true." string | function
A message to show if the assertion fails.

`#assert.that-not(test, message: "Test returned true, should be false.")`

Asserts that test is **false**.

Argument
test boolean
Assertion to test.

Argument
message: "Test returned true, should be false." string | function
A message to show if the assertion fails.

`#assert.eq(a, b, message: (...) => ...)`

Asserts that two values are equal. See `assert.eq`.

Argument
a any
First value.

Argument
b any
Second value.

Argument
message: (...) => ... string | function
A message to show if the assertion fails.

`#assert.ne(a, b, message: (...) => ...)`

Asserts that two values are not equal. See `assert.ne`.

Argument

2.3.1 Assertions

a any
First value.

Argument
b any
Second value.

Argument
message: (...) => ... string | function
A message to show if the assertion fails.

#assert.neq()

Alias for **#ne()**

#assert.not-none(..values, message: (...) => ...)

Asserts that not one of **values** is **none**. Positional and named arguments are tested if provided. For named key-value pairs the value is tested.

Argument
..values any
The values to test.

Argument
message: (...) => ... string | function
A message to show if the assertion fails.

#assert.any(..values, value, message: (...) => ...)

Assert that **value** is any one of **values**.

Tests

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

Argument
value any
Value to compare.

Argument
message: (...) => ... string | function
A message to show if the assertion fails.

#assert.not-any(..values, value, message: (...) => ...)

Assert that **value** is not any one of **values**.

Argument

2.3.1 Assertions

`..values`

any

A set of values to compare value to.

Argument

`value`

any

Value to compare.

Argument

`message: (...) => ...`

string | function

A message to show if the assertion 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: (...) => ...`

string | function

A message to show if the assertion 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: (...) => ...`

string | function

A message to show if the assertion 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: `(...) => ...`

string | function

A message to show if the assertion fails.

#assert.none-of-type(`t`, `..values`, message: `(...) => ...`)

Assert that none of the values are of type `t`.

Argument

`t`

string

The type to test against.

Argument

`..values`

any

Values to test.

Argument

message: `(...) => ...`

string | function

A message to show if the assertion fails.

#assert.not-empty(`value`, message: `(...) => ...`)

Assert that `value` is not *empty*.

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

Argument

`value`

any

The value to test.

Argument

message: `(...) => ...`

string | function

A message to show if the assertion fails.

#assert.has-pos(`n: none`, `args`, message: `(...) => ...`)

Assert that `args` has positional arguments.

If `n` is a value greater zero, exactly `n` positional arguments are required. Otherwise, at least one argument is required.

2.3.1 Assertions

```
1 #let add(..args) = {  
2   assert.has-pos(args)  
3   return args.pos().fold(0, (s, v) => s+v)  
4 }
```

Argument

n: none

integer | none

The mandatory number of positional arguments or none.

Argument

args

arguments

The arguments to test.

Argument

message: (...) => ...

string | function

A message to show if the assertion fails.

#assert.no-pos(args, message: (...) => ...)

Assert that args has no positional arguments.

```
1 #let new-dict(..args) = {  
2   assert.no-pos(args)  
3   return args.named()  
4 }
```

Argument

args

arguments

The arguments to test.

Argument

message: (...) => ...

string | function

A message to show if the assertion fails.

#assert.has-named(names: none, strict: "false", args, message: (...) => ...)

Assert that args has named arguments.

If n is a value greater zero, exactly n named arguments are required. Otherwise, at least one argument is required.

Argument

names: none

array | none

An array with required keys or none.

Argument

strict: "false"

boolean

If true, only keys in names are allowed.

Argument
 args arguments
 The arguments to test.

Argument
 message: (...) => ... string | function
 A message to show if the assertion fails.

#assert.no-named(args, message: (...) => ...)
 Assert that args has no named arguments.

Argument
 args arguments
 The arguments to test.

Argument
 message: (...) => ... string | function
 A message to show if the assertion fails.

#assert.new(test, message: "")
 Creates a new assertion from test.

The new assertion will take a any number of values and pass them to test. test should return a boolean.

```
1 #let assert-numeric = assert.new(is.num)
2
3 #let diameter(radius) = {
4   assert-numeric(radius)
5   return 2*radius
6 }
```

8.6 4

Argument
 test function
 A test function: (.. any) => boolean

II.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.

II.4.1. Command reference

2.4.1 Element helpers

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

#get.dict(..dicts) → dictionary

Create a new dictionary from (

```
sequence(  
  label: <arg-body>,  
  children: (  
    raw(text: "[", block: false, lang: none),  
    styled(  
      child: raw(text: "values", block: false, lang: none),  
      ..,  
    ),  
    raw(text: "]", block: false, lang: none),  
  ),  
)  
).
```

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)
```

—Argument—

`..dicts`

any

Values to merge into the dictionary.

#get.dict-merge(..dicts) → dictionary

Recursivley merges the passed in dictionaries.

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

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

—Argument—

`..dicts`

dictionary

Dictionaries to merge.

#get.args(args, prefix: "") → dictionary

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

2.4.1 Element helpers

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 only 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 as a fallback.

If a `prefix` is specified, only keys with that prefix will be extracted from `args`. The resulting dictionary will have all keys with the prefix removed, though.

```
1 #let my-func( ..options, title ) = block(  
2   ..get.args(options)(  
3     "spacing", "above", "below",  
4     width:100%  
5   )  
6 ) [  
7   #text(..get.args(options, prefix:"text-")(  
8     fill:black, size:0.8em  
9   ), title)  
10 ]  
11  
12 #my-func(  
13   width: 50%,  
14   text-fill: red, text-size: 1.2em  
15 )[#lorem(5)]
```

Argument

`args`

arguments

Argument of a function.

Argument

`prefix: ""`

string

A prefix for the argument keys to extract.

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

Recursively extracts the text content of `element`.

If `element` has children, all child elements are converted to text and joined with `sep`.

- `element` (any)
- `sep` (string, content)

-> string

#get.stroke-paint(stroke, default: "black") → color

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

Compared to `stroke.paint`, this function will return a color for any possible stroke definition (length, dictionary ...).

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

Argument

`stroke`

length | color | dictionary | stroke

2.4.1 Element helpers

The stroke value.

Argument

default: "black"

color

A default color to use.

#get.stroke-thickness(stroke, default: "1pt") → length

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

Compared to stroke.thickness, this function will return a thickness for any possible stroke definition (length, dictionary ...).

Argument

stroke

length | color | dictionary | stroke

The stroke value.

Argument

default: "1pt"

length

A default thickness to use.

#get.stroke-dict(stroke, ..overrides) → dictionary

Converts stroke into a dictionary.

The dictionary will always have the keys thickness, paint, dash, cap and join. If stroke is a dictionary itself, all key/value-pairs are copied to the resulting stroke. Any named arguments in overrides will override the previous values:

```
#let stroke = get.stroke-dict(2pt + red, cap:"square")
```

Argument

stroke

length | color | dictionary | stroke

A stroke value.

Argument

..overrides

any

Overrides for the stroke.

#get.inset-at(direction, inset, default: "0pt") → length

Returns the inset (or outset) in a given direction, ascertained from inset.

Argument

direction

string | alignment

The direction to get.

Argument

inset

length | dictionary

2.4.1 Element helpers

The inset value.

Argument

default: "0pt"

length

A default value.

#get.inset-dict(inset, ..overrides) → dictionary

Creates a dictionary usable as an inset (or outset) argument.

The resulting dictionary is guaranteed to have the keys `top`, `left`, `bottom` and `right`. If `inset` is a dictionary itself, all key/value-pairs are copied to the resulting inset. Any named arguments in `overrides` will override the previous values.

Argument

inset

length | dictionary

The base inset value.

Argument

..overrides

any

Overrides for the inset.

#get.x-align(align, default: left) → alignment

Returns the alignment along the x-axis from `align`.

If none is present, `default` is returned.

```
get.x-align(top + center) // center
```

Argument

align

alignment | 2d alignment

The alignment to get the x-alignment from.

Argument

default: left

alignment

A default alignment.

#get.y-align(align, default: top) → alignment

Returns the alignment along the y-axis from `align`.

If none is present, `default` is returned.

```
get.y-align(top + center) // top
```

Argument

align

alignment | 2d alignment

The alignment to get the y-alignment from.

Argument

default: `top``alignment`

A default alignment.

II.5. Math functions

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

Some functions to complement the native `calc` module.

II.5.1. Command reference

`#clamp()``#lerp()``#map()`

`#math.minmax(a, b) → integer | float | length | relative length | fraction | ratio`

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)
```

Works with any comparable type.

Argument

`a``integer | float | length | relative length | fraction | ratio`

First value.

Argument

`b``integer | float | length | relative length | fraction | ratio`

Second value.

`#math.clamp(min, max, value) → any`

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)
```

Works with any comparable type.

Argument

`min``integer | float | length | relative length | fraction | ratio`

Maximum for value.

Argument

2.5.1 Math functions

value integer | float | length | relative length | fraction | ratio

The value to clamp.

`#math.lerp(min, max, t) → integer | float | length | relative length | fraction | ratio`

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 integer | float | length | relative length | fraction | ratio

Minimum for value.

Argument

max integer | float | length | relative length | fraction | ratio

Maximum for value.

Argument

t float

Interpolation parameter .

`#math.map(min, max, range-min, range-max, value) → integer | float | length | relative length | fraction | ratio`

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))
```

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

Argument

min integer | float | length | relative length | fraction | ratio

Maximum of the initial interval.

Argument

range-min integer | float | length | relative length | fraction | ratio

Maximum of the target interval.

Argument

value integer | float | length | relative length | fraction | ratio

The value to map.

II.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:

- | | | |
|--------------------------|----------------------|------------------------|
| • <code>numbering</code> | • <code>raw</code> | • <code>terms</code> |
| • <code>align</code> | • <code>table</code> | • <code>grid</code> |
| • <code>type</code> | • <code>list</code> | • <code>stack</code> |
| • <code>label</code> | • <code>enum</code> | • <code>columns</code> |
| • <code>text</code> | | |

Part III.

Index

A

#a 3, 5
 #all-of-type 8, 16
 #any 5, 15
 #any-type 8, 16
 #args 20
 #arr 7
 #as-arr 13
 #aut 3, 5

B

#bool 7

C

#clamp 24
 #color 7
 #content 7

D

#dict 6, 20
 #dict-merge 20

E

#elem 8
 #empty 5
 #eq 3, 4, 14

H

#has 6
 #has-named 18
 #has-pos 17

I

#if-any 11
 #if-arg 12
 #if-auto 11
 #if-empty 12
 #if-false 10
 #if-none 10
 #if-not-any 12
 #if-true 9
 #inset-at 22
 #inset-dict 23
 #is-auto 3
 #is-none 3

L

#label 7
 #lerp 25
 #loc 7

M

#map 25
 #minmax 24

N

#n 3, 4
 #ne 13, 14
 #neg 3
 #neq 4, 15
 #new 19
 #no-named 19
 #no-pos 18
 #non 3, 4
 #none-of-type 8, 17
 #not-a 5
 #not-any 6, 15
 #not-any-type 16
 #not-auto 5
 #not-empty 5, 17
 #not-n 4
 #not-none 4, 15
 #numbering 26

O

#one-not-none 5

S

#same-type 8
 #sequence 9
 #stroke 7
 #stroke-dict 22
 #stroke-paint 21
 #stroke-thickness 22

T

#text 21
 #that 13, 14
 #that-not 14
 #type 6

X

#x-align 23

Y

#y-align 23