



Tools For Typst

v0.4.1

2024-12-11

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	7
II.2.1. Command reference	7
II.3. Assertions	11
II.3.1. Command reference	12
II.4. Element helpers	17
II.4.1. Command reference	18
II.5. Math functions	22
II.5.1. Command reference	22
II.6. Alias functions	24

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:0.4.1": 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/0.4.1
```

In either case, make sure the files are placed in a subfolder with the correct version number: `t4t/0.4.1`

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

```
#import "@local/t4t:0.4.1": automaton
```

I.2. Manual

The manual is created using [Tidy](https://github.com/Mc-Zen/tidy)³ with the [Mantys](https://github.com/jneug/typst-mantys)⁴ template.

[Tidy](https://github.com/jneug/typst-tools4typst) will be loaded from the package repository while [Mantys](https://github.com/jneug/typst-mantys) needs to be installed manually into the local package repository. Refer to the [Mantys](https://github.com/jneug/typst-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": test
```

These functions provide shortcuts to common tests like `#test.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:

```
// check all values for none
if some-array.any(is-none) {
  ...
}

// find first not none value
let x = (none, none, 5, none).find(not-none)

// find position of a value
let pos-bar = args.pos().position(test.eq.with("|"))
```

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

```
#import "@preview/t4t:0.1.4": is-none, is-auto
```

The `t4t.is` submodule still has these tests, but under different names (`#test.n()` and `#test.non()` for `none` and `#test.a()` and `#test.aut()` for `auto`).

II.1.1. Command reference

<code>#all-of-type()</code>	<code>#is-elem()</code>	<code>#neq()</code>
<code>#any()</code>	<code>#is-empty()</code>	<code>#none-of-type()</code>
<code>#any-type()</code>	<code>#is-sequence()</code>	<code>#not-any()</code>
<code>#eq()</code>	<code>#is-type()</code>	<code>#one-not-none()</code>
<code>#has()</code>	<code>#neg()</code>	<code>#same-type()</code>

`#test.neg({test})` → `function`

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

Can be used to create negations of tests like:

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

2.1 Test functions

Argument

<test>

function | bool

Test to negate.

#test.eq(<compare>, <value>) → bool

Tests if values <compare> and <value> are equal.

Argument

<compare>

any

first value

Argument

<value>

any

second value

#test.neq(<compare>, <value>) → bool

Tests if <compare> and <value> are not equal.

Argument

<compare>

any

First value.

Argument

<value>

any

Second value.

#test.is-empty(<value>) → bool

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

#test.one-not-none(..<values>) → bool

Tests, if at least one value in <values> is not equal to *none*.

Useful for checking mutiple optional arguments for a valid value:

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

Argument

..<<values>

any

Values to test.

#test.any(..<compare>, <value>) → bool

2.1 Test functions

Tests, if any value of `..<compare>` is equal to `<value>`.

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

Argument	
<code><value></code>	any
value to test	

`#test.not-any(..<compare>, <value>)` → `bool`

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

Argument	
<code>..<code><compare></code></code>	any
values to compare to	

Argument	
<code><value></code>	any
value to test	

`#test.has(..<keys>, <value>)` → `bool`

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	
<code>..<code><keys></code></code>	any
keys or values to look for	

Argument	
<code><value></code>	any
value to test	

`#test.is-type(t, <value>)`

Tests if `<value>` is of type `t`.

Argument	
<code><t></code>	str
name of the type	

Argument	
<code><value></code>	any
value to test	

`#test.any-type(..<types>, <value>)`

Tests if types `<value>` is any one of types.

2.1 Test functions

Argument	
.. <types>	str
type names to check against	

Argument	
<value>	any
value to test	

#test.same-type(..(values))

Tests if all passed in values have the same type.

Argument	
.. <values>	any
Values to test.	

#test.all-of-type(<t>, ..(values))

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

Argument	
<t>	str
type to test against	

Argument	
.. <values>	any
Values to test.	

#test.none-of-type(<t>, ..(values))

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

Argument	
<t>	str
type to test against	

Argument	
.. <values>	any
Values to test.	

#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:

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

Argument	
<func>	function

element function

Argument

<value>

any

value to test

#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:

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

II.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>, <value>, <def>: none, <do>: none)

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>

bool

A test result.

Argument

<value>

any

2.2 Default values

The value to test.

Argument

`<def>: none`

any

The default value.

Argument

`<do>: none`

function

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

#def.if-false(`<test>`, `<value>`, `<def>: none`, `<do>: none`)

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>`

bool

A test result.

Argument

`<value>`

any

The value to test.

Argument

`<def>: none`

any

The default value.

Argument

`<do>: none`

function

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

#def.if-none(`<value>`, `<def>: none`, `<do>: none`)

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

`<value>`

any

The value to test.

Argument

`<def>: none`

any

The default value.

Argument

`<do>: none`

function

2.2 Default values

Post-processor for $\langle \text{value} \rangle$: (any) => any

#def.if-auto($\langle \text{value} \rangle$, $\langle \text{def} \rangle$: none, $\langle \text{do} \rangle$: none)

Returns $\langle \text{default} \rangle$ if $\langle \text{value} \rangle$ is auto, $\langle \text{value} \rangle$ otherwise.

If $\langle \text{value} \rangle$ is not auto and $\langle \text{do} \rangle$ is set to a function, $\langle \text{value} \rangle$ is passed to $\langle \text{do} \rangle$, before being returned.

Argument
 $\langle \text{value} \rangle$ any
The value to test.

Argument
 $\langle \text{def} \rangle$: none any
A default value.

Argument
 $\langle \text{do} \rangle$: none function
Post-processor for $\langle \text{value} \rangle$: (any) => any

#def.if-any($\langle \text{value} \rangle$, $\langle \text{compare} \rangle$, $\langle \text{def} \rangle$: none, $\langle \text{do} \rangle$: none)

Returns $\langle \text{def} \rangle$ if $\langle \text{value} \rangle$ is equal to any value in $\langle \text{compare} \rangle$, $\langle \text{value} \rangle$ otherwise.

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

If $\langle \text{value} \rangle$ is in $\langle \text{compare} \rangle$ and $\langle \text{do} \rangle$ is set to a function, $\langle \text{value} \rangle$ is passed to $\langle \text{do} \rangle$, before being returned.

Argument
 $\langle \text{value} \rangle$ any
value to test

Argument
 $\langle \text{compare} \rangle$ any
list of values to compare $\langle \text{value} \rangle$ to

Argument
 $\langle \text{def} \rangle$: none any
The default value.

Argument
 $\langle \text{do} \rangle$: none function
Post-processor for $\langle \text{value} \rangle$: (any) => any

#def.if-not-any($\langle \text{value} \rangle$, $\langle \text{compare} \rangle$, $\langle \text{def} \rangle$: none, $\langle \text{do} \rangle$: none)

2.2 Default values

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
`<value>` any
value to test

Argument
..`<compare>` any
list of values to compare `<value>` to

Argument
`<def>`: none any
The default value.

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

```
#def.if-empty(<value>, <def>: none, <do>: none)
```

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 `t4t.is-empty()`. See there for an explanation of *empty*.

Argument
`<value>` any
value to test

Argument
`<def>`: none any
The default value.

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

```
#def.if-arg(<key>, <def>: none, <do>: none, <args>)
```

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

2.2 Default values

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

Argument	
<code><key></code>	any
key to look for	
Argument	
<code><def></code> : none	any
The default value.	
Argument	
<code><do></code> : none	function
Post-processor for <code><value></code> : (any) => any	
Argument	
<code><args></code>	arguments
arguments to test	

`#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

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

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

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

Argument

`<test>` bool

Assertion to test.

Argument

`<message>: "Test returned false, should be true."` str | 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>` bool

Assertion to test.

Argument

`<message>: "Test returned true, should be false."` str | function

A message to show if the assertion fails.

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

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

Argument

`<a>` any

First value.

Argument

`` any

Second value.

Argument

`<message>: (...) => ...` str | function

A message to show if the assertion fails.

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

2.3 Assertions

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

Argument	
<code><a></code>	any
First value.	
Argument	
<code></code>	any
Second value.	
Argument	
<code><message>: (...) => ...</code>	str function
A message to show if the assertion fails.	

#`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	
<code>..<code><values></code></code>	any
The values to test.	
Argument	
<code><message>: (...) => ...</code>	str function
A message to show if the assertion fails.	

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

Assert that `<value>` is any one of `<values>`.

Tests

Argument	
<code>..<code><values></code></code>	any
A set of values to compare <code><value></code> to.	
Argument	
<code><value></code>	any
Value to compare.	
Argument	
<code><message>: (...) => ...</code>	str function
A message to show if the assertion fails.	

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

Assert that `<value>` is not any one of `<values>`.

2.3 Assertions

Argument

..**<values>**

any

A set of values to compare value to.

Argument

<value>

any

Value to compare.

Argument

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

str | function

A message to show if the assertion fails.

#assert.any-type(..**<types>**, **<value>**, **<message>: (...)** => ...)

Assert that **<value>**s type is any one of **<types>**.

Argument

..**<types>**

str

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

Argument

<value>

any

Value to compare.

Argument

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

str | function

A message to show if the assertion fails.

#assert.not-any-type(..**<types>**, **<value>**, **<message>: (...)** => ...)

Assert that **<value>**s type is not any one of **<types>**.

Argument

..**<types>**

str

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

Argument

<value>

any

Value to compare.

Argument

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

str | 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>**.

2.3 Assertions

Argument	
<code><t></code>	str
The type to test against.	

Argument	
<code>..<code><values></code></code>	any
Values to test.	

Argument	
<code><message>: (...) => ...</code>	str 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	
<code><t></code>	str
The type to test against.	

Argument	
<code>..<code><values></code></code>	any
Values to test.	

Argument	
<code><message>: (...) => ...</code>	str function
A message to show if the assertion fails.	

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

Assert that `<value>` is not *empty*.

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

Argument	
<code><value></code>	any
The value to test.	

Argument	
<code><message>: (...) => ...</code>	str 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 Assertions

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

Argument

⟨n⟩: **none**

int | none

The mandatory number of positional arguments or **none**.

Argument

⟨args⟩

arguments

The arguments to test.

Argument

⟨message⟩: (...) => ...

str | function

A message to show if the assertion fails.

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

Assert that ⟨args⟩ has no positional arguments.

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

Argument

⟨args⟩

arguments

The arguments to test.

Argument

⟨message⟩: (...) => ...

str | 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**

bool

If **true**, only keys in ⟨names⟩ are allowed.

2.3 Assertions

Argument

<args>

arguments

The arguments to test.

Argument

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

str | 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>: (...) => ...

str | function

A message to show if the assertion fails.

#assert.new(<test>, <message>: "")

Creates a new assertion from test.

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

```
#let assert-numeric = assert.new(t4t.is-num)

#let diameter(radius) = {
  assert-numeric(radius)
  return 2*radius
}
```

8.6 4

Argument

<test>

function

A test function: (..any) => bool

#neq

Alias for #ne()

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

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

#get.dict(..{dicts}) → **dictionary**

Create a new dictionary from (

```
sequence(
  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

..

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

..

dictionary

Dictionaries to merge.

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

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 only

2.4 Element helpers

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.

```
#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)]
```

Argument

`<args>`

arguments

Argument of a function.

Argument

`<prefix>: ""`

str

A prefix for the argument keys to extract.

`#get.text(<element>, <sep>: "")` → `str`

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)

`#get.stroke-paint(<stroke>, <default>: luma(0%))` → `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

The stroke value.

Argument

`<default>: luma(0%)`

color

2.4 Element helpers

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>

str | alignment

The direction to get.

Argument

<inset>

length | dictionary

The inset value.

Argument

<default>: "0pt"

length

2.4 Element helpers

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()      #map()
#lerp()       #minmax()
```

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

First value.

Argument

<*b*> `int` | `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*> `int` | `float` | `length` | `relative length` | `fraction` | `ratio`

Maximum for value.

2.5 Math functions

Argument

<value> `int` | `float` | `length` | `relative length` | `fraction` | `ratio`

The value to clamp.

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

Minimum for value.

Argument

<max> `int` | `float` | `length` | `relative length` | `fraction` | `ratio`

Maximum for value.

Argument

<t> `float`

Interpolation parameter .

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

Maximum of the initial interval.

Argument

<range-min> `int` | `float` | `length` | `relative length` | `fraction` | `ratio`

Maximum of the target interval.

Argument

<value>

int | 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:

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

The following functions have aliases right now:

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

Part III.

Index

A

`#a` 3
`#all-of-type` 6, 14
`#any` 4, 13
`#any-type` 5, 14
`#args` 18
`#as-arr` 11
`#aut` 3

C

`#clamp` 22

D

`#dict` 18
`#dict-merge` 18

E

`#eq` 3, 4, 11, 12

H

`#has` 5
`#has-named` 16
`#has-pos` 15

I

`#if-any` 9
`#if-arg` 10
`#if-auto` 9
`#if-empty` 10
`#if-false` 8
`#if-none` 8
`#if-not-any` 9
`#if-true` 7
`#inset-at` 20
`#inset-dict` 21
`#is-auto` 3
`#is-elem` 6
`#is-empty` 4
`#is-none` 3
`#is-sequence` 7
`#is-type` 5

L

`#lerp` 23

M

`#map` 23
`#minmax` 22

N

`#n` 3
`#ne` 11, 12
`#neg` 3
`#neq` 4
`#new` 17
`#no-named` 17
`#no-pos` 16
`#non` 3
`#none-of-type` 6, 15
`#not-any` 5, 13
`#not-any-type` 14
`#not-empty` 15
`#not-none` 13
`#numbering` 24

O

`#one-not-none` 4

S

`#same-type` 6
`#stroke-dict` 20
`#stroke-paint` 19
`#stroke-thickness` 20

T

`#text` 19
`#that` 11, 12
`#that-not` 12

X

`#x-align` 21

Y

`#y-align` 21