# t4t

#### **Tools For Typst**

v0.3.0 2023-08-24 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)
I.2. Manual2
II. Module reference
II.1. Test functions 3
II.1.1. Command reference 3
II.2. Default values 10
II.2.1. Command reference 11
II.3. Assertions 15
II.3.1. Command reference 15
II.4. Element helpers23
II.4.1. Command reference 23
II.5. Math functions28
II.5.1. Command reference 28
II.6. Alias functions
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:0.2.0:*
```

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 t4t/0.2.0
```

In either case, make sure the files are placed in a subfolder with the correct version number: t4t/0.2.0

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

```
#import "@local/t4t:0.2.0:*
```

#### I.2. Manual

The manual is created using TIDY<sup>3</sup> with the MANTYS<sup>4</sup> 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.

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

<sup>&</sup>lt;sup>2</sup>https://github.com/typst/packages#local-packages

<sup>3</sup>https://github.com/Mc-Zen/tidy

<sup>4</sup>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: <code>#is-none()</code> and <code>#is-auto()</code>. Since keywords can't be used as function names, the <code>is</code> module can't define a function like <code>is.none()</code>. Therefore the functions <code>#is-none()</code> and <code>#is-auto()</code> are provided in the base module of <code>t4t</code>:

```
#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

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

```
\#is.neg(test) \rightarrow function
```

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

Can be used to create negations of tests like:



Alias for n().

 $\#is.not-none(..values) \rightarrow boolean$ 

# Tests if none of values is equal to none. ..values any Values to test. #is.not-n() Alias for not-none() #is.one-not-none(..values) $\rightarrow$ boolean Tests, if at least one value in values is not equal to none. Useful for checking mutliple optional arguments for a valid value: #if is.one-not-none(..args.pos()) [ #args.pos().find(is.not-none) ..values any Values to test. $\#is.a(..values) \rightarrow boolean$ Tests if any one of values is equal to auto. ..values any Values to test. #is.aut() Alias for a() $\#is.not-auto(..values) \rightarrow boolean$ Tests if none of values is equal to auto. ..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. value any value to test #is.not-empty(value) → boolean Tests, if value is not *empty*. See empty() for an explanation what *empty* means. value any value to test  $\#is.any(..compare, value) \rightarrow boolean$ Tests, if value is not *empty*. See empty() for an explanation what *empty* means. value any value to test  $\#is.not-any(..compare, value) \rightarrow boolean$ Tests if value is not equals to any one of the other passed in values. ..compare any



#### $\#is.has(..keys, value) \rightarrow boolean$

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.



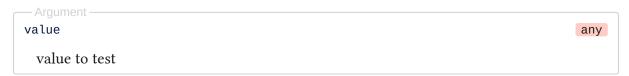
#### #is.type(t, value)

Tests if value is of type t.



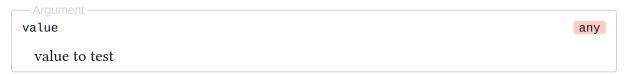
#### #is.dict(value)

Tests if value is of type dictionary.



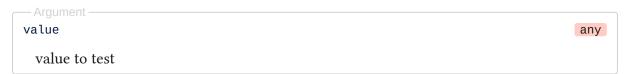
#### #is.arr(value)

Tests if value is of type array.



#### #is.content(value)

Tests if value is of type content.



#### #is.label(value)

Tests if value is of type label.



#### #is.color(value)

Tests if value is of type color.

```
value value to test
```

#### #is.stroke(value)

Tests if value is of type stroke.

```
value value to test
```

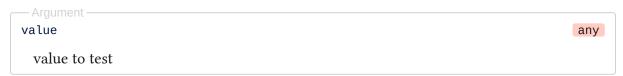
#### #is.loc(value)

Tests if value is of type location.



#### #is.bool(value)

Tests if value is of type boolean.



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

Tests if types value is any one of types.



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

Tests if all passed in values have the same type.



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

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



Values to test.

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

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



#### #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)
```

```
Func function element function
```

```
value 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.

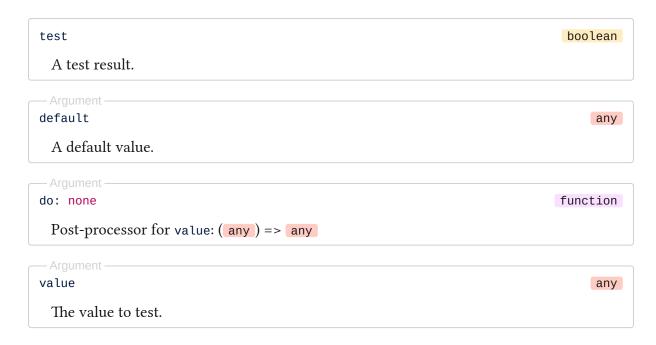


```
#def.if-false(test, default, do: "none", value)
```

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 — Argument —
```



#### #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.



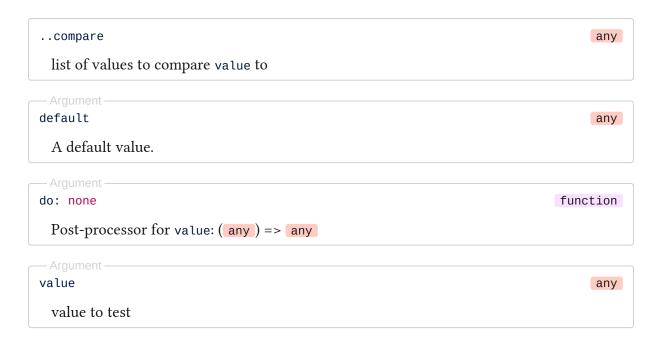
#### #def.if-auto(default, do: "none", value)

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.



```
do: none
                                                                               function
   Post-processor for value: (any) => any
 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
                     // default
    1pt,
                     // value
    thickness
  )
  If value is in compare and do is set to a function, value is passed to do, before being returned.
  ..compare
                                                                                    any
   list of values to compare value to
 default
                                                                                    any
   A default value.
                                                                               function
 do: none
   Post-processor for value: (any) => any
 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
                                  // default
    left,
                                  // value
    position
  )
  If value is in compare and do is set to a function, value is passed to do, before being returned.
```



#### #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*.



#### #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.



#### #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

```
#all-of-type()
                              #ne()
                                                          #not-any()
                              #neq()
  #any()
                                                           #not-any-type()
  #any-type()
                              #new()
                                                           #not-empty()
  #eq()
                              #no-named()
                                                           #not-none()
                              #no-pos()
  #has-named()
                                                           #that()
                              #none-of-type()
  #has-pos()
                                                           #that-not()
#assert.that(test, message: "none")
  Asserts that test is true. See assert.
                                                                               boolean
 test
   Assertion to test.
                                                                     string | function
 message: none
   A message to show if the assertion fails.
#assert.that-not(test, message: "none")
  Asserts that test is false.
                                                                               boolean
 test
   Assertion to test.
                                                                     string | function
 message: none
   A message to show if the assertion fails.
#assert.eq(a, b, message: (..a) => "Values should be the same, got " + repr(a.pos()))
  Asserts that two values are equal. See assert.eq.
                                                                                   any
   First value.
 b
                                                                                   any
   Second value.
                                                                     string | function
 message: (..) \Rightarrow ..
```

A message to show if the assertion fails.

```
#assert.ne(a, b, message: (..a) => "Values should not be the same, got " +
repr(a.pos()))
  Asserts that two values are not equal. See assert.ne.
 a
                                                                                    any
   First value.
 b
                                                                                    any
   Second value.
 message: (..) \Rightarrow ..
                                                                      string | function
   A message to show if the assertion fails.
#assert.neq()
  Alias for ne()
#assert.not-none(..values, message: (..a) => "Values should not be none. Got " +
repr(a))
  Asserts that not one of values is none. Positional and named arguments are tested if pro-
  vided. For named key-value pairs the value is tested.
  ..values
                                                                                    any
   The values to test.
                                                                      string | function
 message: (..) \Rightarrow ..
   A message to show if the assertion fails.
#assert.any(..values, value, message: (..a) => "Allowed values: " +
repr(a.pos().slice(1)) + ". Got " + repr(a.pos().first()))
  Assert that value is any one of values.
```

#### Tests

```
..values
                                                                                   any
   A set of values to compare value to.
 value
                                                                                   any
   Value to compare.
                                                                     string function
 message: (..) \Rightarrow ..
   A message to show if the assertion fails.
#assert.not-any(..values, value, message: (..a) => "Disallowed values: " +
repr(a.pos().slice(1)) + ". Got " + repr(a.pos().first()))
  Assert that value is not any one of values.
  ..values
                                                                                   any
   A set of values to compare value to.
 value
                                                                                   any
   Value to compare.
 message: (..) \Rightarrow ..
                                                                     string function
   A message to show if the assertion fails.
#assert.any-type(..types, value, message: (..a) => "\nAllowed types: "
+ repr(a.pos().slice(1)) + ".\nGot " + repr(a.pos().first()) + " (" +
type(a.pos().first()) + ")")
  Assert that values type is any one of types.
  ..types
                                                                                string
   A set of types to compare the type of value to.
 value
                                                                                   any
   Value to compare.
```

```
string | function
 message: (..) \Rightarrow ..
   A message to show if the assertion fails.
#assert.not-any-type(..types, value, message: (..a) => "\nDisallowed types:
" + repr(a.pos().slice(1)) + ".\nGot " + repr(a.pos().first()) + " (" +
type(a.pos().first()) + ")")
  Assert that values type is not any one of types.
 ..types
                                                                                string
   A set of types to compare the type of value to.
 value
                                                                                   any
   Value to compare.
 message: (..) \Rightarrow ..
                                                                     string function
   A message to show if the assertion fails.
\#assert.all-of-type(t, ...values, message: (...a) => "\nValues need to be of
type " + repr(a.pos().first()) + ".\nGot " + repr(a.pos().slice(1)) + " / " +
repr(a.pos().slice(1).map(type)))
  Assert that the types of all values are equal to t.
                                                                                string
 t
   The type to test against.
 ..values
                                                                                   anv
   Values to test.
                                                                     string function
 message: (..) \Rightarrow ...
   A message to show if the assertion fails.
```

```
\#assert.none-of-type(t, ...values, message: (...a) => "\nValues may not be of
type " + repr(a.pos().first()) + ".\nGot " + repr(a.pos().slice(1)) + " / " +
repr(a.pos().slice(1).map(type)))
  Assert that none of the values are of type t.
                                                                               string
 t
   The type to test against.
 ..values
                                                                                  any
   Values to test.
 message: (..) \Rightarrow ..
                                                                    string | function
   A message to show if the assertion fails.
#assert.not-empty(value, message: (v, ..a) => {
    "Value may not be empty. Got " + repr(v)
  })
  Assert that value is not empty.
  Depends on is.empty(). See there for an explanation of empty.
 value
                                                                                  any
   The value to test.
 message: (..) => ..
                                                                    string function
   A message to show if the assertion fails.
#assert.has-pos(n: "none", args, message: (n:none, ..a) => {
  if n == none {
    "At least one positional argument required."
  } else {
    "Exactly " + str(n) +" positional arguments required, got " + repr(a.pos())
})
  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.

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

n: none integer | none

The mandatory number of positional arguments or none.

```
args arguments

The arguments to test.
```

```
message: (..) => .. string | function

A message to show if the assertion fails.
```

#assert.no-pos(args, message: (..a) => "Unexpected positional arguments: " + repr(a))
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.
```

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

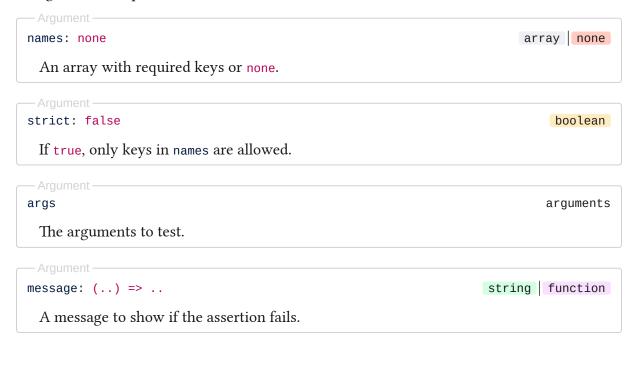
A message to show if the assertion fails.
```

```
#assert.has-named(names: "none", strict: "false", args, message: (..a) => {
    let names = a.named().at("names", default:())
    if names == () {
        "Missing named arguments."
    } else {
        let named = a.named()
        let keys = named.keys()
        names = names.filter((k) => k != "names" and k not in keys)
        "Missing named arguments: " + names.join(", ")
```

```
})
```

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.



```
#assert.no-named(args, message: (..a) => "Unexpected named arguments: " +
repr(a.named()))
```

Assert that args has no named arguments.

```
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 }
```

#### 8.64

```
Argument test

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

```
#args() #inset-at() #stroke-paint()
#dict() #inset-dict() #stroke-thickness()
#dict-merge() #stroke-dict() #text()
```

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

Values to merge into the dictionary.
```

#### $\#get.dict-merge(..dicts) \rightarrow dictionary$

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: """") $\rightarrow$ dictionary

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

#### 2.4.1 Element helpers

```
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
The stroke value.

Argument
default: rgb("#000000")
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 ...).

```
stroke | length | color | dictionary | stroke
```

#### 2.4.1 Element helpers

The stroke value.

Argument

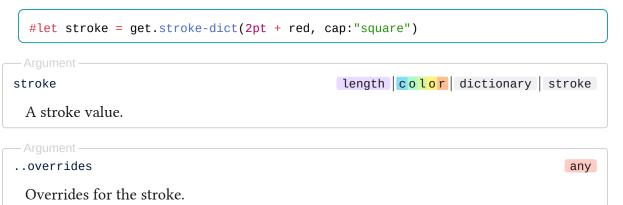
default: 1pt

A default thickness to use.

#### $\#get.stroke-dict(stroke, ..overrides) \rightarrow dictionary$

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:



#### $\#get.inset-at(direction, inset, default: "Opt") \rightarrow length$

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

Argument

The direction to get.

Argument
inset

The inset value.

Argument

default: Opt

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.



```
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.

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

First value.
```

```
b integer | float | length | relative length | fraction | ratio |
Second value.
```

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

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

#### 2.5.1 Math functions

# Maximum for value. —Argument— value integer | float | length | relative length | fraction | ratio

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

Calculates the linear interpolation of t between min and max.

The value to clamp.

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))
```

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

Minimum for value.
```

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

Maximum for value.
```

```
t 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.

#### 2.5.1 Math functions

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

Maximum of the initial interval.
```

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

Maximum of the target interval.
```

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

The following functions have aliases right now:

numbering

raw

• terms

align

• table

• grid

type

• list

stack

• label

enum

• columns

text

## Part III.

# Index

Α	L	Υ
#a 3, 5	#label 8	#y-align 27
#all-of-type 9, 19	#lerp 29	
#any 6, 17	#loc 8	
#any-type 9, 18		
#args 24	M	
#arr 8	#map	
#as-arr 15	#minmax 28	
#aut 3, 5	N	
В	#n 3, 4	
#bool 9	#ne 15, 17	
	#neg 3	
C	#neq 4, 17	
#clamp 28	#new 22	
#color 8	#no-named 22	
#content 8	#no-pos 21	
D	#non 3, 4	
#dict 7, 23	#none-of-type 10, 20	
#dict-merge 24	#not-a6	
#ulot merge	#not-any 6, 18	
E	#not-any-type 19 #not-auto 5	
#elem 10	#not-empty 6, 20	
#empty 6	#not-n5	
#eq 3, 4, 16	#not-none 5, 17	
н	#numbering 30	
#has 7		
#has-named 21	0	
#has-pos 20	#one-not-none5	
,	S	
1	#same-type 9	
#if-any 13	#sequence 10	
#if-arg 14	#stroke 8	
#if-auto 12	#stroke-dict 26	
#if-empty 14	#stroke-paint 25	
#if-false 11	#stroke-thickness25	
#if-none	<b>-</b>	
#if-not-any 13	T	
#if-true 11 #inset-at 26	#text	
#inset-dict 27	#that 15, 16	
#is-auto 3	#that-not 16	
#is-none	#type 7	
	X	
	#x-align 27	