The "t4t" package

Tools For Typst

v0.4.2 2025-02-27 MIT

An utility package for typst package authors.

Jonas Neugebauer

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

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 GitHub[1] and unpack the archive into your system dependent local repository folder[3] 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[5] with the Mantys[7] template.

[1] https://github.com/jneug/typst-tools4typst[2]
[2] https://github.com/jneug/typst-tools4typst
[3] https://github.com/typst/packages#local-packages[4]
[4] https://github.com/typst/packages#local-packages
[5] https://github.com/Mc-Zen/tidy[6]
[6] https://github.com/Mc-Zen/tidy
[7] https://github.com/jneug/typst-mantys[8]
[8] https://github.com/jneug/typst-mantys

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.

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:

```
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(not-none)
8
9  // find position of a value
10 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`:

```
1 #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

#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.any(⟨value⟩, ..⟨compare⟩) → **bool**

Tests, if any value of ..⟨compare⟩ is equal to ⟨value⟩.

See cmd:test.is-empty for an explanation what *empty* means.

Argument
⟨value⟩ — any

value to test

#test.any-type(⟨value⟩, ..⟨types⟩)

Tests if types ⟨value⟩ is any one of `types`.

Argument
⟨value⟩ — any

value to test

Argument
..⟨types⟩ — str

type names to check against

#test.eq(⟨compare⟩, ⟨value⟩) → **bool**

Tests if values ⟨compare⟩ and ⟨value⟩ are equal.

Argument
⟨compare⟩ — any

first value

Argument
⟨value⟩ any

second value

#test.has((value), ..(keys)) → **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
⟨value⟩ any

value to test

Argument
..⟨keys⟩ any

keys or values to look for

#test.is-elem((func), (value))

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

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

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

Argument
⟨func⟩ function

element function

Argument
⟨value⟩ any

value to test

#test.is-empty((value)) → **bool**

Tests, if ⟨value⟩ is *empty*.

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

Argument

⟨value⟩ `any`

value to test

`#test.is-sequence(⟨value⟩)`

Tests if ⟨value⟩ is a `sequence` of content.

`#test.is-type(⟨t⟩, ⟨value⟩)`

Tests if ⟨value⟩ is of type `t`.

Argument

⟨t⟩ `str`

name of the type

Argument

⟨value⟩ `any`

value to test

`#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-raw = test.neg(test.is-raw)
```

Argument

⟨test⟩ `function` | `bool`

Test to negate.

`#test.neq(⟨compare⟩, ⟨value⟩)` → `bool`

Tests if ⟨compare⟩ and ⟨value⟩ are not equal.

Argument

⟨compare⟩ `any`

First value.

Argument

⟨value⟩ `any`

Second value.

`#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.not-any(⟨value⟩, ..⟨compare⟩) → `bool`

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

Argument

⟨value⟩ `any`

value to test

Argument

..⟨compare⟩ `any`

values to compare to

#test.one-not-none(..⟨values⟩) → `bool`

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

Useful for checking mutliple 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.same-type(..⟨values⟩)

Tests if all passed in values have the same type.

Argument

..⟨values⟩ `any`

Values to test.

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

```
#def.as-arr              #def.if-auto             #def.if-none
#def.if-any              #def.if-empty            #def.if-not-any
#def.if-arg              #def.if-false            #def.if-true
```

`#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(", ")
```

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

Returns `⟨def⟩` if `⟨value⟩` is equal to any value in `compare`, `⟨value⟩` otherwise.

```
#def.if-any(
  thickness,      // value
  none, auto,     // ..compare
```

```
4    def: 1pt,       // default
5  )
```

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
⟨def⟩: none any

The default value.

Argument
⟨do⟩: none function

Post-processor for ⟨value⟩: (any)→ any

Argument
..⟨compare⟩ any

list of values to compare ⟨value⟩ to

#def.if-arg(⟨key⟩, ⟨args⟩, ⟨def⟩: none, ⟨do⟩: none)

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
⟨key⟩ any

key to look for

Argument
⟨args⟩ arguments

arguments to test

Argument
⟨def⟩: none any

The default value.

Argument
⟨do⟩: none `function`

Post-processor for ⟨value⟩: (any)→ any

#def.if-auto(⟨value⟩, ⟨def⟩: none, ⟨do⟩: none)

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
⟨value⟩ `any`

The value to test.

Argument
⟨def⟩: none `any`

A 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 `test.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-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

Post-processor for ⟨value⟩: (any)→ any

#def.if-not-any(⟨value⟩, ⟨def⟩: none, ⟨do⟩: none, ..⟨compare⟩)

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
⟨def⟩: none `any`

The default value.

Argument
⟨do⟩: none `function`

Post-processor for ⟨value⟩: (`any`)→ `any`

Argument
..⟨compare⟩ `any`

list of values to compare ⟨value⟩ to

#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`

The value to test.

Argument
⟨def⟩: none `any`

The default value.

Argument

⟨do⟩: none `function`

Post-processor for ⟨value⟩: (any)→ any

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

#assert.all-of-type	#assert.ne	#assert.not-any-type
#assert.any	#assert.new	#assert.not-empty
#assert.any-type	#assert.no-named	#assert.not-none
#assert.eq	#assert.no-pos	#assert.that
#assert.has-named	#assert.none-of-type	#assert.that-not
#assert.has-pos	#assert.not-any	

```
#assert.all-of-type(⟨t⟩, ⟨message⟩: (..a) => (
    "Values need to be of type "
      + repr(a.pos().first())
      + ". Got "
      + repr(a.pos().slice(1))
      + " / "
      + repr(a.pos().slice(1).map(type))
  ), ..⟨values⟩)
```

Assert that the types of all ⟨values⟩ are equal to ⟨t⟩.

Argument

⟨t⟩ `str`

The type to test against.

Argument

```
(message): (..a) => (
    "Values need to be of type "
      + repr(a.pos().first())
      + ". Got "
      + repr(a.pos().slice(1))
      + " / "
      + repr(a.pos().slice(1).map(type))
  )
```

str | function

A message to show if the assertion fails.

Argument

..(values) any

Values to test.

```
#assert.any((value), (message): (..a) => "Value should be one of " +
repr(a.pos().slice(1)) + ". Got " + repr(a.pos().first()), ..(values))
```

Assert that (value) is any one of (values).

Tests

Argument

(value) any

Value to compare.

Argument

```
(message): (..a) => "Value should be one of " + repr(a.pos().slice(1))
+ ". Got " + repr(a.pos().first())
```

str | function

A message to show if the assertion fails.

Argument

..(values) any

A set of values to compare (value) to.

```
#assert.any-type((value), (message): (..a) => (
    "Value should have any type of "
      + repr(a.pos().slice(1))
      + ". Got "
      + repr(a.pos().first())
      + " ("
      + type(a.pos().first())
      + ")"
  ), ..(types))
```

Assert that ⟨value⟩s type is any one of ⟨types⟩.

Argument

⟨value⟩ `any`

Value to compare.

Argument

⟨message⟩: (..a) => (
    "Value should have any type of "
      + repr(a.pos().slice(1))
      + ". Got "
      + repr(a.pos().first())
      + " ("
      + type(a.pos().first())
      + ")"
) `str` | `function`

A message to show if the assertion fails.

Argument

..⟨types⟩ `str`

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

```
#assert.eq((a), (b), (message): (a, b) => "Value " + repr(a) + " was not equal to " + repr(b))
```

Asserts that two values are equal.

Argument

⟨a⟩ `any`

First value.

Argument

⟨b⟩ `any`

Second value.

Argument

⟨message⟩: (a, b) => "Value " + repr(a) + " was not equal to " + repr(b)

str | function

A message to show if the assertion fails.

```
#assert.has-named(⟨args⟩, ⟨names⟩: none, ⟨strict⟩: false, ⟨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.

Argument

⟨args⟩ arguments

The arguments to test.

Argument

⟨names⟩: none array | none

An array with required keys or none.

Argument

⟨strict⟩: false bool

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

Argument

```
⟨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(", ")
    }
  }
```

`str` | `function`

A message to show if the assertion fails.

```
#assert.has-pos((args), (n): none, (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 }
```

Argument

`(args)` `arguments`

The arguments to test.

Argument

`(n): none` `int` | `none`

The mandatory number of positional arguments or `none`.

Argument

```
(message): (n: none, ..a) => {
    if n == none {
      "At least one positional argument required."
```

```
    } else {
       "Exactly " + str(n) + " positional arguments required, got "
+ repr(a.pos())
    }
  }
```

str | function

A message to show if the assertion fails.

`#assert.ne((a), (b), (message): (a, b) => "Value " + repr(a) + " was equal to " + repr(b))`

Asserts that two values are not equal.

Argument

`(a)` any

First value.

Argument

`(b)` any

Second value.

Argument

`(message): (a, b) => "Value " + repr(a) + " was equal to " + repr(b)`

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 `bool`.

```
1 #let assert-numeric = assert.new(t4t.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`)→ `bool`

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

Assert that ⟨args⟩ has no named arguments.

Argument

⟨args⟩ `arguments`

The arguments to test.

Argument

⟨message⟩: `(..a) => "Unexpected named arguments: " + repr(a.named())`
`str` | `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.

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

Argument

⟨args⟩ `arguments`

The arguments to test.

Argument

⟨message⟩: `(..a) => "Unexpected positional arguments: " + repr(a)` `str` | `function`

A message to show if the assertion fails.

```
#assert.none-of-type(⟨t⟩, ⟨message⟩: (..a) => (
    "Values may not be of type "
      + repr(a.pos().first())
      + ". Got "
      + repr(a.pos().slice(1))
      + " / "
      + repr(a.pos().slice(1).map(type))
  ), ..⟨values⟩)
```

Assert that none of the ⟨values⟩ are of type ⟨t⟩.

Argument

⟨t⟩ `str`

The type to test against.

Argument

⟨message⟩: (..a) => (
    "Values may not be of type "
      + repr(a.pos().first())
      + ". Got "
      + repr(a.pos().slice(1))
      + " / "
      + repr(a.pos().slice(1).map(type))
) `str` | `function`

A message to show if the assertion fails.

Argument

..⟨values⟩ `any`

Values to test.

```
#assert.not-any(⟨value⟩, ⟨message⟩: (..a) => "Value should not be one of "
+ repr(a.pos().slice(1)) + ". Got " + repr(a.pos().first()), ..⟨values⟩)
```

Assert that ⟨value⟩ is not any one of ⟨values⟩.

Argument

⟨value⟩ `any`

Value to compare.

Argument

⟨message⟩: (..a) => "Value should not be one of " + repr(a.pos().slice(1)) + ". Got " + repr(a.pos().first()) `str` | `function`

A message to show if the assertion fails.

Argument

`..⟨values⟩` `any`

A set of values to compare `value` to.

```
#assert.not-any-type(⟨value⟩, ⟨message⟩: (..a) => (
    "Value should not have any type of "
      + repr(a.pos().slice(1))
      + ". Got "
      + repr(a.pos().first())
      + " ("
      + type(a.pos().first())
      + ")"
  ), ..⟨types⟩)
```

Assert that `⟨value⟩`s type is not any one of `⟨types⟩`.

Argument

`⟨value⟩` `any`

Value to compare.

Argument

```
⟨message⟩: (..a) => (
    "Value should not have any type of "
      + repr(a.pos().slice(1))
      + ". Got "
      + repr(a.pos().first())
      + " ("
      + type(a.pos().first())
      + ")"
  )
```

`str` | `function`

A message to show if the assertion fails.

Argument

`..⟨types⟩` `str`

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

```
#assert.not-empty(⟨value⟩, ⟨message⟩: (v, ..a) => {
    "Value may not be empty. Got " + repr(v)
  })
```

Assert that `⟨value⟩` is not *empty*.

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

Argument

⟨value⟩ any

The value to test.

Argument

⟨message⟩: (v, ..a) => {
    "Value may not be empty. Got " + repr(v)
  } str | function

A message to show if the assertion fails.

#assert.not-none((message): (..a) => "Values should not be none. Got " + repr(a), ..(values))

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

⟨message⟩: (..a) => "Values should not be none. Got " + repr(a) str | function

A message to show if the assertion fails.

Argument

..⟨values⟩ any

The values to test.

#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.

`#neq`

Alias for cmd:assert.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

```
#get.args                #get.inset-dict          #get.text
#get.dict                #get.stroke-dict         #get.x-align
#get.dict-merge          #get.stroke-paint        #get.y-align
#get.inset-at            #get.stroke-thickness
```

`#get.args((args), (prefix): "") → 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(
    ..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.dict(..⟨dicts⟩) → `dictionary`

Create a new dictionary from `(metadata(value: (func: (..) => .., args: (), kind: "barg")),)`.

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[9].

Argument

`..⟨dicts⟩` `dictionary`

Dictionaries to merge.

#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`

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.

[9] https://github.com/johannes-wolf/typst-canvas/

Argument
⟨inset⟩ length | dictionary

The base inset value.

Argument
..⟨overrides⟩ any

Overrides for the inset.

#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.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[10].

Argument
⟨stroke⟩ length | color | dictionary | stroke

The stroke value.

Argument
⟨default⟩: black color

[10] https://github.com/PgBiel/typst-tablex

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.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 (str, content)

`#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

```
#math.clamp                 #math.map
#math.lerp                  #math.minmax
```

`#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`.

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 cmd:math.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.

`#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.

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