Contents

1 Overview 4
2 Getting Started 4
2.1 Usage 4
2.2 Examples 4
3 Basics 4
3.1 Custom Types 4
3.2 The Canvas 4
3.3 Styling 5
4 API 5
4.1 Canvas 5
canvas 5
4.2 Shapes 6
circle 6
circle-through 7
arc 8
arc-through 9
mark 10
line 10
polygon 11
n-star 12
grid 13
content 13
rect 15
bezier 16
bezier-through 17
catmull 17
hobby 18
compound-path 19
merge-path 19
rect-around 20
svg-path 21
4.3 Styling 21
set-style 21
fill 22
stroke 22
register-mark 22
4.4 Grouping 23
hide 23
floating 24
intersections 24
group 25
scope 26
anchor 27
copy-anchors 27
set-ctx 28
get-ctx 28
for-each-anchor 28
on-layer 29
4.5 Transformation 30
set-transform 30
rotate 30
translate 30
scale 31
set-origin 31
move-to 31
set-viewport 32
4.6 Projection 32
ortho 32
on-xy 33
on-xz 34
on-yz 34
4.7 Utility 34
assert-version 34
register-coordinate-resolver 35
4.8 Libraries 35
Angle 35
angle 35
right-angle 37
Tree 37
tree 37
Decorations 39
Path 39
Brace 41
Palette 44
new 44
4.9 Internals 44
Complex 44
re 44
im 45
mul 45
conj 45
dot 45
normsq 46
norm 46
scale 46
unit 46
inv 46
div 47
add 47
sub 47
arg 47
ang 48
Vector 48
as-mat 48
as-vec 48
len 49

add ... 49
sub ... 49
dist ... 49
scale ... 50
div ... 50
neg ... 50
norm ... 50
element-product ... 51
dot ... 51
cross ... 51
angle2 ... 51
angle ... 52
lerp ... 52
Matrix ... 52
ident ... 52
diag ... 53
dim ... 53
column ... 53
round ... 53
transform-translate ... 54
transform-shear-x ... 54
transform-shear-z ... 54
transform-scale ... 54
transform-rotate-dir ... 55
transform-rotate-x ... 55
transform-rotate-y ... 55
transform-rotate-z ... 55
transform-rotate-xz ... 55
transform-rotate-ypr ... 56
transform-rotate-xyz ... 56
mul-mat ... 56
mul4x4-vec3 ... 57
mul-vec ... 57
inverse ... 57
Coordinate ... 58
resolve-system ... 58
resolve ... 58
Styles ... 58
resolve ... 58
merge ... 60
Process ... 60
element ... 60
many ... 60
Drawable ... 61
TAG ... 61
apply-transform ... 61
apply-tags ... 61
filter-tagged ... 61
path ... 62
line-strip ... 62
content ... 63
ellipse ... 63
arc ... 64
Anchor ... 65
border ... 65
setup ... 65
Mark ... 66
check-mark ... 66
process-style ... 67
place-mark-on-path ... 67
place-marks-along-path ... 68
Bezier ... 68
quadratic-point ... 68
quadratic-derivative ... 69
cubic-point ... 69
cubic-derivative ... 69
to-abc ... 70
quadratic-through-3points ... 71
quadratic-to-cubic ... 71
cubic-through-3points ... 71
split ... 72
cubic-arclen ... 72
cubic-shorten-linear ... 72
cubic-t-for-distance ... 73
cubic-shorten ... 73
cubic-extrema ... 74
cubic-aabb ... 75
catmull-to-cubic ... 75
line-cubic-intersections ... 75
AABB ... 76
aabb ... 76
mid ... 76
corner-points ... 76
size ... 77
padded ... 77
Hobby ... 77
hobby-to-cubic-open ... 77
hobby-to-cubic-closed ... 78
hobby-to-cubic ... 78
Intersection ... 79
line-line ... 79
line-cubic ... 80
line-path ... 80
path-path ... 80
Path Util ... 81
make-subpath ... 81
first-subpath-closed ... 81
first-subpath-start ... 81

subpath-start 81
subpath-end 82
last-subpath-end 82
bounds 82
segment-lengths 82
length 82
point-at 83
shorten-to 83
normalize 84
Util 84
float-epsilon 84
float-eq 84
apply-transform 85
revert-transform 85
line-pt 85
line-normal 86
circle-arclen 86
ellipse-point 86
calculate-circle-center-3pt 86
resolve-number 87
map-dict 87
resolve-radius 87
min 87
max 88
merge-dictionary 88
measure 88
as-padding-dict 88
as-corner-radius-dict 89
sort-points-by-distance 89
resolve-stroke 90
assert-body 90

1 Overview

CeTZ, ein Typst Zeichenpaket, is a drawing package for Typst. Its API is similar to Processing but with relative coordinates and anchors from TikZ. You also won't have to worry about accidentally drawing over other content as the canvas will automatically resize. And remember: up is positive!

2 Getting Started

2.1 Usage

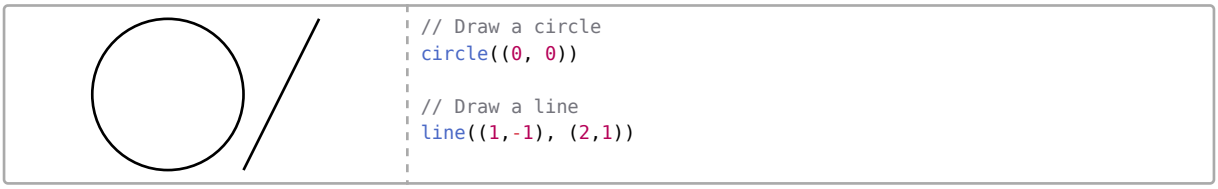
This is the minimal starting point in a `.typ` file:

```
#import "@preview/cetz:0.4.2"
#cetz.canvas({
  import cetz.draw: *
  ...
})
```

Note that draw functions are imported inside the scope of the canvas block. This is recommended as some draw functions override Typst's functions such as `line` .

2.2 Examples

From this point on only the code inside the canvas block will be shown in examples unless specified otherwise.



```
// Draw a circle
circle((0, 0))

// Draw a line
line((1,-1), (2,1))
```

3 Basics

The following chapters are about the basic and core concepts of CeTZ. They are recommended reading for basic usage.

3.1 Custom Types

Many CeTZ functions expect data in certain formats which we will call types. Note that these are actually made up of Typst primitives.

`coordinate` A position on the canvas specified by any coordinate system. See Coordinate Systems.

`number` Any of `float`, `int` or `length`

`style` Represents options passed to draw functions that affect how elements are drawn. They are normally taken in the form of named arguments to the draw functions or sometimes can be a dictionary for a single argument.

3.2 The Canvas

The canvas function is what handles all of the logic and processing in order to produce drawings. It's usually called with a code block `{ ... }` as argument. The content of the curly braces is the body of the canvas. Import all the draw functions you need at the top of the body:

```
#cetz.canvas({
  import cetz.draw: *
})
```

You can now call the draw functions within the body and they'll produce some graphics! Typst will evaluate the code block and pass the result to the canvas function for rendering.

The canvas does not have typical width and height parameters. Instead its size will grow and shrink to fit the drawn graphic.

By default 1 coordinate unit is 1 cm, this can be changed by setting the `length:` parameter.

3.3 Styling

You can style draw elements by passing the relevant named arguments to their draw functions. All elements that draw something have stroke and fill styling unless said otherwise.

fill `color` or `none` (default: `none`)

How to fill the drawn element.

stroke `none` or `auto` or `length` or `color` or `dictionary` or `stroke` (default: `black`)

How to stroke the border or the path of the draw element. See Typst's line documentation for more details.

4 API

4.1 Canvas

canvas

```
canvas(
  length: length,
  x: number vector,
  y: number vector,
  z: number vector,
  baseline: none number coordinate,
  debug: bool,
  background: none color,
  stroke: none stroke,
  padding: none number array dictionary,
  body none array element
) ⟶ content
```

Sets up a canvas for drawing on.

Parameters

- **`length`** `length`

  Used to specify what 1 coordinate unit is. Note that ratios are no longer supported! You can wrap the canvas into a `layout(ly => canvas(length: ly.width * <ratio>, ...))`.

- **`baseline`** `none` `number` `coordinate`

  Specifies the coordinate to use as the baseline. Setting this the canvas behaves like a `box` element instead of a `block`.

- **`body`** `none` `array` `element`

  A code block in which functions from the `draw` module have been called.

- **`background`** `none` `color`

  A color to be used for the background of the canvas.

- **`stroke`** `none` `stroke`

  Stroke style to apply to the canvas top-level element (box or block)
- **`padding`** `none` `number` `array` `dictionary`

  How much padding to add to the canvas. `none` applies no padding. A number applies padding to all sides equally. A dictionary applies padding following Typst's `pad` function: https://typst.app/docs/reference/layout/pad/. An array follows CSS like padding: `(y, x)`, `(top, x, bottom)` or `(top, right, bottom, left)`.
- **`x`** `number` `vector`

  Sets up the x vector of the coordinate system to `(x, 0, 0)` or to the given vector.
- **`y`** `number` `vector`

  Sets up the y vector of the coordinate system to `(0, y, 0)` or to the given vector.
- **`z`** `number` `vector`

  Sets up the z vector of the coordinate system to `(0, 0, z)` or to the given vector.
- **`debug`** `bool`

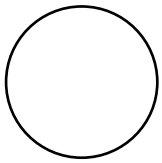
  Shows the bounding boxes of each element when `true`.

4.2 Shapes

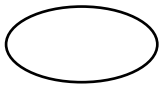
circle

```
circle(
  ..points-style coordinate style,
  name: none str,
  anchor: none str
)
```

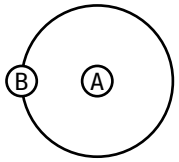
Draws a circle or ellipse.



```
// Draw a circle with center (0, 0)
circle((0, 0))
```



```
// Draw an ellipse
circle((2, 0), radius: (1, 0.5))
```




```
let (a, b) = ((2, 1), (1, 1))

// Draw a circle with its center at (2, 1), going
// through point (1, 1)
circle(a, b)

// Show both points
set-style(content: (frame: "circle", padding: 1pt, fill: white))
content(a, [A]); content(b, [B])
```

Styling

Root: `circle`

Anchors

Supports border and path anchors. The `"center"` anchor is the default.

Parameters

- **`..points-style`** `coordinate` `style`

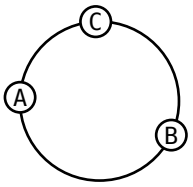
  The position to place the circle on. If given two coordinates, the distance between them is used as radius. If given a single coordinate, the radius can be set via the `radius` (style) argument.
- **`name`** `none` `str`
- **`anchor`** `none` `str`
- **`radius`** `number` `array`

  A number that defines the size of the circle's radius. Can also be set to a tuple of two numbers to define the radii of an ellipse, the first number is the `x` radius and the second is the `y` radius.

circle-through

```
circle-through(
  a coordinate,
  b coordinate,
  c coordinate,
  name: none str,
  anchor: none str,
  ..style style
)
```

Draws a circle through three coordinates.




```typst
let (a, b, c) = ((0, 0), (2, -0.5), (1, 1))

// Draw a circle through 3 points
circle-through(a, b, c, name: "c")

// Show the points
set-style(content: (frame: "circle", padding: 1pt, fill: white))
content(a, [A]); content(b, [B]); content(c, [C])
```

Styling

Root: `circle`

`circle-through` has the same styling as `circle` except for `radius` as the circle's radius is calculated by the given coordinates.

Anchors

Supports the same anchors as `circle` as well as:

a Coordinate a
b Coordinate b
c Coordinate c

Parameters

- **`a`** `coordinate`

  Coordinate a.
- **`b`** `coordinate`

  Coordinate b.
- **`c`** `coordinate`

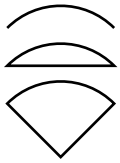
  Coordinate c.
- **`name`** `none` `str`

- **anchor** `none` `str`
- **..style** `style`

arc

```
arc(
  position coordinate,
  start: auto angle,
  stop: auto angle,
  delta: auto angle,
  name: none str,
  anchor: none str,
  ..style style
)
```

Draws a circular segment.



```
arc((0,0), start: 45deg, stop: 135deg)
arc((0,-0.5), start: 45deg, delta: 90deg, mode: "CLOSE")
arc((0,-1), stop: 135deg, delta: 90deg, mode: "PIE")
```

Note that two of the three angle arguments (`start`, `stop` and `delta`) must be set. The current position `()` gets updated to the arc's end coordinate (anchor `arc-end`).

Styling

Root: `arc`

Anchors

Supports border and path anchors.

arc-start The position at which the arc's curve starts, this is the default.
arc-end The position of the arc's curve end.
arc-center The midpoint of the arc's curve.
center The center of the arc, this position changes depending on if the arc is closed or not.
chord-center Center of chord of the arc drawn between the start and end point.
origin The origin of the arc's circle.

Parameters

- **position** `coordinate`

  Position to place the arc at.
- **start** `auto` `angle`

  The angle at which the arc should start. Remember that `0deg` points directly towards the right and `90deg` points up.
- **stop** `auto` `angle`

  The angle at which the arc should stop.
- **delta** `auto` `angle`

  The change in angle away start or stop.
- **name** `none` `str`
- **anchor** `none` `str`
- **..style** `style`

- **`radius`** `number` `array`

  The radius of the arc. An elliptical arc can be created by passing a tuple of numbers where the first element is the x radius and the second element is the y radius.
- **`mode`** `str`

  The options are: `"OPEN"` no additional lines are drawn so just the arc is shown; `"CLOSE"` a line is drawn from the start to the end of the arc creating a circular segment; `"PIE"` lines are drawn from the start and end of the arc to the origin creating a circular sector.
- **`update-position`** `bool`

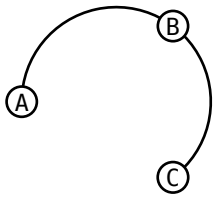
  Update the current canvas position to the arc's end point (anchor `"arc-end"`). This overrides the default of `true`, that allows chaining of (arc) elements.

arc-through

```
arc-through(
  a coordinate,
  b coordinate,
  c coordinate,
  name: none str,
  ..style style
)
```

Draws an arc that passes through three points a, b and c.

Note that all three points must not lie on a straight line, otherwise the function fails.




```
let (a, b, c) = ((0, 1), (2, 2), (2, 0))

// Draw an arc through 3 points
arc-through(a, b, c)

// Show the points
set-style(content: (frame: "circle", padding: 1pt, fill: white))
content(a, [A]); content(b, [B]); content(c, [C])
```

Styling

Root: `arc`

Uses the same styling as `arc`.

Anchors

For anchors see `arc`.

Parameters

- **`a`** `coordinate`

  Start position of the arc
- **`b`** `coordinate`

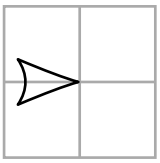
  Position the arc passes through
- **`c`** `coordinate`

  End position of the arc
- **`name`** `none` `str`
- **`..style`** `style`

mark

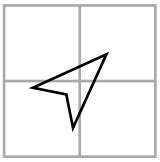
```
mark(
  from coordinate,
  to coordinate angle,
  ..style str style
)
```

Draws a single mark pointing towards a target coordinate.



```
// Show a grid
grid((-1, -1), (1, 1), stroke: gray)

// Draw a mark with its tip at (0, 0) pointing to (1, 0)
mark((0, 0), (1, 0), symbol: ")>", scale: 4)
```



```
// Show a grid
grid((-1, -1), (1, 1), stroke: gray)

// Draw a mark with its center at (0, 0) pointing to (1, 1)
mark((0, 0), (1, 1), symbol: ">>", anchor: "center", scale: 5)
```

Note: To place a mark centered at the first coodinate (`from`) use the marks `anchor: "center"` style.

Styling

Root: `mark`

You can directly use the styling from mark styling.

Parameters

- **`from`** `coordinate`

  The position to place the mark.
- **`to`** `coordinate` `angle`

  The position or angle the mark should point towards.
- **`..style`** `str` `style`

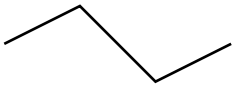
  If the third positional argument is of type string, it is treated as mark name (e.g. `">"`) and overrules style keys such as `mark.symbol` or `mark.end`

line

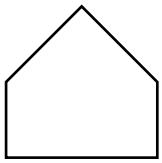
```
line(
  ..pts-style coordinate style,
  close: bool,
  name: none str
)
```

Draws a line, more than two points can be given to create a line-strip.

```
// Draw a line between two points
line((0, 0), (1.5, 1))
```

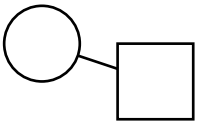


```
// Draw a line between more than two points
line((0, 0), (1, 0.5), (2, -0.5), (3, 0))
```



```
// Draw a polygon using `close: true`
line((0, 0), (0, 1), (1, 2), (2, 1), (2,0), close: true)
```

If the first or last coordinates are given as the name of an element, that has a `"default"` anchor, the intersection of that element's border and a line from the first or last two coordinates given is used as coordinate. This is useful to span a line between the borders of two elements.



```
circle((1,2), radius: .5, name: "a")
rect((2,1), (rel: (1,1)), name: "b")
line("a", "b")
```

Styling

Root: `line`

Supports mark styling.

Anchors

Supports path anchors.

centroid The centroid anchor is calculated for *closed non self-intersecting* polygons if all vertices share the same z value.

Parameters

- **`..pts-style`** `coordinate` `style`

  Positional two or more coordinates to draw lines between. Accepts style key-value pairs.

- **`close`** `bool`

  If true, the line-strip gets closed to form a polygon

- **`name`** `none` `str`

polygon

```
polygon(
  origin coordinate,
  sides int,
  angle: angle,
  name: none str,
  anchor: ,
  ..style
)
```

Draws a regular polygon.

```
set-style(polygon: (radius: 0.65))

polygon((0, 0), 3, angle: 90deg)
polygon((1.5,0), 5)
polygon((3, 0), 7)
```

Styling

Root: `polygon`

Parameters

- **`origin`** `coordinate`
  Coordinate to draw the polygon at
- **`sides`** `int`
  Number of sides of the polygon (>= 3)
- **`angle`** `angle`
  Angle angle to rotate the polygon arround its origin
- **`name`** `none` `str`
- **`radius`** `number`
  Radius of the polygon

n-star

```
n-star(
  origin coordinate,
  sides int,
  angle: angle,
  name: none str,
  anchor: ,
  ..style
)
```

Draws a n-pointed star.

```
set-style(n-star: (radius: 0.65))

n-star((0, 0), 5)

// An 8-pointed star, rotated
n-star((1.5, 0), 8, angle: 11.25deg)

// A 6-pointed star showing its inner hexagon
n-star((3, 0), 6, show-inner: true)
```

Styling

Root: nstar

Parameters

- **`origin`** `coordinate`
  Coordinate to draw the star's center at.
- **`sides`** `int`
  Number of points of the star (>= 3).
- **`angle`** `angle`
  Angle to rotate the star around its origin.
- **`name`** `none` `str`
  An optional name to identify the shape.
- **`radius`** `number`
  The radius of the star's outer points.
- **`inner-radius`** `number` `ratio`
  The radius (if of type ratio, relative to the outer radius) of the star's inner points of the star's inner points.

- **`show-inner`** `bool`

  If true, also draws the inner polygon connecting the star's inner points.
- **`fill`** `color` `gradient`

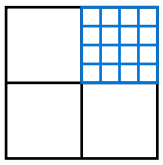
  The fill color for the star.
- **`stroke`** `color` `thickness` `...`

  The stroke for the star and the inner polygon.

grid

```
grid(
  from coordinate,
  to coordinate,
  name: none str,
  ..style style
)
```

Draws a grid between two coordinates



```
// Draw a grid
grid((0,0), (2,2))

// Draw a smaller blue grid
grid((1,1), (2,2), stroke: blue, step: .25)
```

Styling

Root: `grid`

Anchors

Supports border anchors.

Parameters

- **`from`** `coordinate`

  The top left of the grid
- **`to`** `coordinate`

  The bottom right of the grid
- **`name`** `none` `str`
- **`..style`** `style`
- **`step`** `number` `array` `dictionary`

  Distance between grid lines. A distance of 1 means to draw a grid line every 1 length units in x- and y-direction. If given a dictionary with `x` and `y` keys or a tuple, the step is set per axis.
- **`shift`** `number` `array` `dictionary`

  Offset of the grid lines. Supports an array of the form `(x, y)` or a dictionary of the form `(x: <number>, y: <number>)`.
- **`help-lines`** `bool`

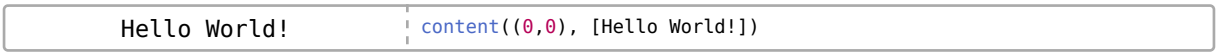
  If true, force the stroke style to `gray + 0.2pt`

content

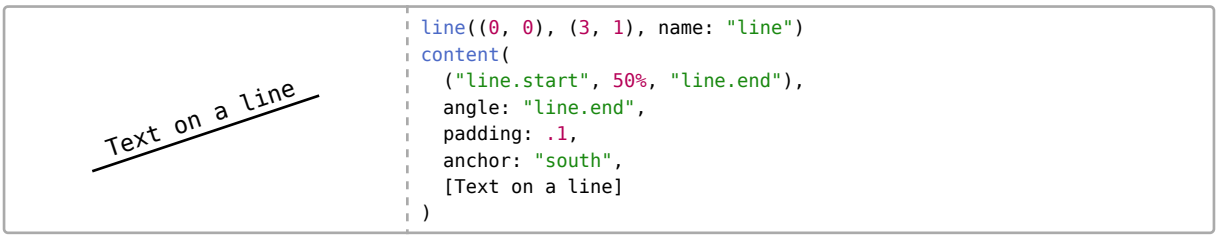
```
content(
  ..args-style coordinate content style,
  angle: angle coordinate,
```

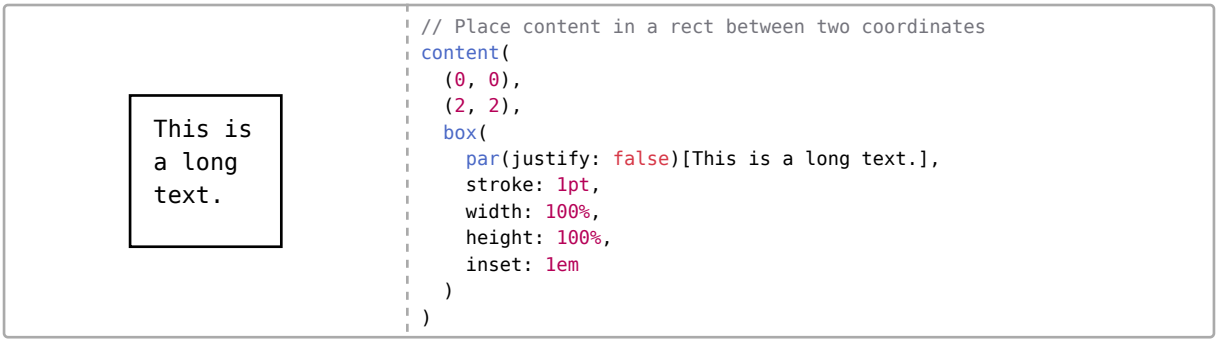
```
  anchor: none str,
  name: none str
)
```

Positions Typst content in the canvas. Note that the content itself is not transformed only its position is.



To put text on a line you can let the function calculate the angle between its position and a second coordinate by passing it to `angle`:





Styling

Root: `content`

Anchors

Supports border anchors, the default anchor is set to **center**.
mid Content center, from baseline to top bounds
mid-east Content center extended to the east
mid-west Content center extended to the west
base Horizontally centered baseline of the content
base-east Baseline height extended to the east
base-west Baseline height extended to the west
text Position at the content start on the baseline of the content

Parameters

- **`..args-style`** `coordinate` `content` `style`

  When one coordinate is given as a positional argument, the content will be placed at that position. When two coordinates are given as positional arguments, the content will be placed inside a rectangle between the two positions. All named arguments are styling and any additional positional arguments will panic.

- **`angle`** `angle` `coordinate`

  Rotates the content by the given angle. A coordinate can be given to rotate the content by the angle between it and the first coordinate given in `args`. This effectively points the right hand side

of the content towards the coordinate. This currently exists because Typst's rotate function does not change the width and height of content.

- **`anchor`** `none` `str`
- **`name`** `none` `str`
- **`padding`** `number` `dictionary`

  Sets the spacing around content. Can be a single number to set padding on all sides or a dictionary to specify each side specifically. The dictionary follows Typst's `pad` function: https://typst.app/docs/reference/layout/pad/
- **`frame`** `str` `none`

  Sets the frame style. Can be {{none}}, `"rect"` or `"circle"` and inherits the `stroke` and `fill` style.
- **`auto-scale`** `bool`

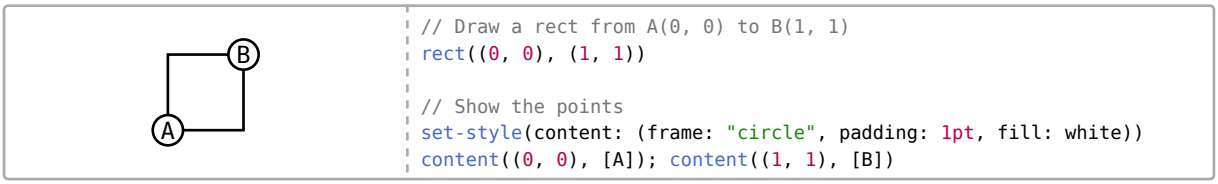
  If `true`, apply current canvas scaling to the content. Defaults to `false`.
- **`wrap`** `function` `none`

  A function to apply the content body to. Must return content. Example: `text.with(red)` to wrap every content element in a `text(red, <body>)` element.

rect

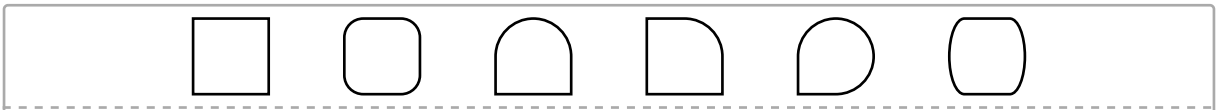
```
rect(
  a coordinate,
  b coordinate,
  name: none str,
  anchor: none str,
  ..style style
)
```

Draws a rectangle between two coordinates.




```typ
// Draw a rect from A(0, 0) to B(1, 1)
rect((0, 0), (1, 1))

// Show the points
set-style(content: (frame: "circle", padding: 1pt, fill: white))
content((0, 0), [A]); content((1, 1), [B])
```



```typ
rect((0,0), (rel: (1,1)), radius: 0)
rect((2,0), (rel: (1,1)), radius: 25%)
rect((4,0), (rel: (1,1)), radius: (north: 50%))
rect((6,0), (rel: (1,1)), radius: (north-east: 50%))
rect((8,0), (rel: (1,1)), radius: (south-west: 0, rest: 50%))
rect((10,0), (rel: (1,1)), radius: (rest: (20%, 50%)))
```

Styling

Root: `rect`

Anchors

Supports border and path anchors. It's default is the `"center"` anchor.

Parameters

- **`a`** `coordinate`

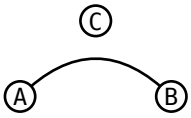
  Coordinate of the bottom left corner of the rectangle.

- **`b`** `coordinate`
  Coordinate of the top right corner of the rectangle. You can draw a rectangle with a specified width and height by using relative coordinates for this parameter `(rel: (width, height))`.
- **`name`** `none` `str`
- **`anchor`** `none` `str`
- **`..style`** `style`
- **`radius`** `number` `ratio` `dictionary`
  The rectangle's corner radius. If set to a single number, that radius is applied to all four corners of the rectangle. If passed a dictionary you can set the radii per corner. The following keys support either a `number`, `ratio` or an array of `number` or `ratio` for specifying a different x- and y-radius: `north`, `east`, `south`, `west`, `north-west`, `north-east`, `south-west` and `south-east`. To set a default value for remaining corners, the `rest` key can be used. Ratio values are relative to the rectangle's width and height.

bezier

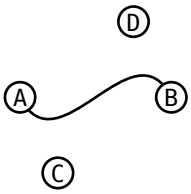
```
bezier(
  start coordinate,
  end coordinate,
  ..ctrl-style coordinate style,
  name: none str
)
```

Draws a quadratic or cubic bezier curve




```typ
let (a, b, c) = ((0, 0), (2, 0), (1, 1))
bezier(a, b, c)

set-style(content: (frame: "circle", padding: 1pt, fill: white))
content(a, [A]); content(b, [B]); content(c, [C])
```




```typ
let (a, b, c, d) = ((0, 0), (2, 0), (.5, -1), (1.5, 1))
bezier(a, b, c, d)

set-style(content: (frame: "circle", padding: 1pt, fill: white))
content(a, [A]); content(b, [B]); content(c, [C]); content(d, [D])
```

Styling

Root `bezier`

Supports marks.

Anchors

Supports path anchors.
ctrl-n nth control point where n is an integer starting at 0

Parameters

- **`start`** `coordinate`
  Start position
- **`end`** `coordinate`
  End position (last coordinate)

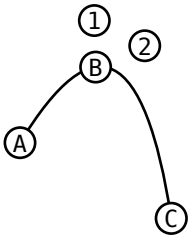
- **`name`** `none` `str`
- **`..ctrl-style`** `coordinate` `style`

  The first two positional arguments are taken as cubic bezier control points, where the first is the start control point and the second is the end control point. One control point can be given for a quadratic bezier curve instead. Named arguments are for styling.

bezier-through

```
bezier-through(
  start coordinate,
  pass-through coordinate,
  end coordinate,
  name: none str,
  ..style style
)
```

Draws a cubic bezier curve through a set of three points. See `bezier` for style and anchor details.




```
let (a, b, c) = ((0, 0), (1, 1), (2, -1))
bezier-through(a, b, c, name: "curve")

// Show the computed control points: 1 and 2
set-style(content: (frame: "circle", padding: 1pt, fill: white))
content(a, [A]); content(b, [B]); content(c, [C])
content("curve.ctrl-1", [2]); content("curve.ctrl-0", [1])
```

Parameters

- **`start`** `coordinate`

  The position to start the curve.
- **`pass-through`** `coordinate`

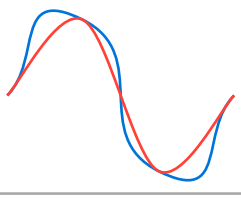
  The position to pass the curve through.
- **`end`** `coordinate`

  The position to end the curve.
- **`name`** `none` `str`
- **`..style`** `style`

catmull

```
catmull(
  ..pts-style coordinate style,
  close: bool,
  name: none str
)
```

Draws a Catmull-Rom curve through a set of points.



```
catmull((0,0), (1,1), (2,-1), (3,0), tension: .4, stroke: blue)
catmull((0,0), (1,1), (2,-1), (3,0), tension: .5, stroke: red)
```

Styling

Root: `catmull`

Supports marks.

Anchors

Supports path anchors.
pt-n The nth given position (0 indexed so "pt-0" is equal to "start")

Parameters

- **`..pts-style`** `coordinate` `style`

  Positional arguments should be coordinates that the curve should pass through. Named arguments are for styling.
- **`close`** `bool`

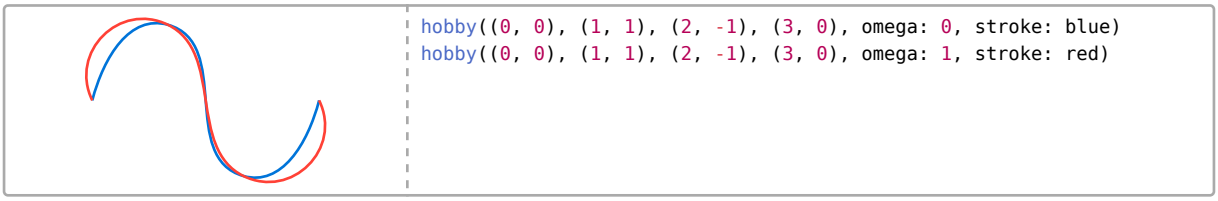
  Closes the curve with a straight line between the start and end of the curve.
- **`name`** `none` `str`
- **`tension`** `float`

  How tight the curve should fit to the points. The higher the tension the less curvy the curve.

hobby

```
hobby(
  ..pts-style coordinate style,
  ta: auto array,
  tb: auto array,
  close: bool,
  name: none str
)
```

Draws a Hobby curve through a set of points.



```
hobby((0, 0), (1, 1), (2, -1), (3, 0), omega: 0, stroke: blue)
hobby((0, 0), (1, 1), (2, -1), (3, 0), omega: 1, stroke: red)
```

Styling

Root `hobby`

Supports marks.

Anchors

Supports path anchors.
pt-n The nth given position (0 indexed, so "pt-0" is equal to "start")

Parameters

- **`..pts-style`** `coordinate` `style`

  Positional arguments are the coordinates to use to draw the curve with, a minimum of two is required. Named arguments are for styling.

- **`tb`** `auto` `array`

  Incoming tension at `pts.at(n+1)` from `pts.at(n)` to `pts.at(n+1)`. The number given must be one less than the number of points.
- **`ta`** `auto` `array`

  Outgoing tension at `pts.at(n)` from `pts.at(n)` to `pts.at(n+1)`. The number given must be one less than the number of points.
- **`close`** `bool`

  Closes the curve with a proper smooth curve between the start and end of the curve.
- **`name`** `none` `str`
- **`omega`** `array`

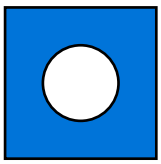
  A tuple of floats that describe how curly the curve should be at each endpoint. When the curl is close to zero, the spline approaches a straight line near the endpoints. When the curl is close to one, it approaches a circular arc.

compound-path

```
compound-path(
  body elements,
  name: none str,
  ..style style
)
```

Create a new path with each element used as sub-paths. This can be used to create paths with holes.

Unlike `merge-path`, this function groups the shapes as sub-paths instead of concattenating them into a single continous path.



```
compound-path({
  rect((-1, -1), (1, 1))
  circle((0, 0), radius: .5)
}, fill: blue, fill-rule: "even-odd")
```

Anchors

centroid Centroid of the *closed and non self-intersecting* shape. Only exists if `close` is true. Supports path anchors and shapes where all vertices share the same z-value.

Parameters

- **`body`** `elements`

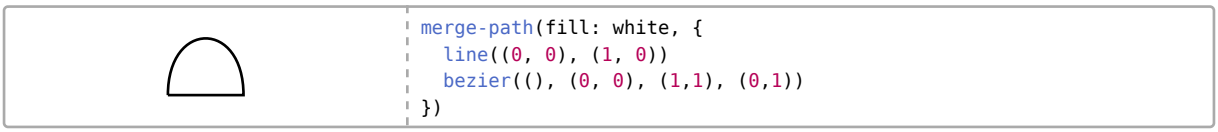
  Elements with paths to be merged together.
- **`name`** `none` `str`
- **`..style`** `style`

merge-path

```
merge-path(
  body elements,
  join: bool,
  ignore-marks: bool,
  ignore-hidden: bool,
  close: bool,
  name: none str,
```

```
  ..style style
)
```

Merges two or more paths by concattenating their elements. Anchors and visual styling, such as `stroke` and `fill`, are not preserved. When an element's path does not start at the same position the previous element's path ended, a straight line is drawn between them so that the final path is continuous. You must then pay attention to the direction in which element paths are drawn.



```
merge-path(fill: white, {
  line((0, 0), (1, 0))
  bezier((), (0, 0), (1,1), (0,1))
})
```

Elements hidden via [hide](../grouping/hide) are ignored.

Anchors

centroid Centroid of the *closed and non self-intersecting* shape. Only exists if `close` is true. Supports path anchors and shapes where all vertices share the same z-value.

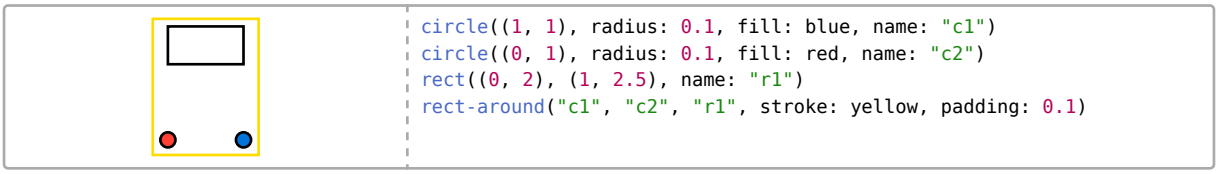
Parameters

- **`body`** `elements`
  Elements with paths to be merged together.
- **`join`** `bool`
  Connect all sup-paths with a straight line
- **`close`** `bool`
  Close the path with a straight line from the start of the path to its end.
- **`ignore-marks`** `bool`
  If true, remove marks from input elements
- **`ignore-hidden`** `bool`
  If true, ignore all hidden elements
- **`name`** `none` `str`
- **`..style`** `style`

rect-around

```
rect-around(
  ..pts-style coordinates style
)
```

Draws an axis aligned bounding box around all given points/elements. Everything else (styling, anchors) is similar to the rect shape.



```
circle((1, 1), radius: 0.1, fill: blue, name: "c1")
circle((0, 1), radius: 0.1, fill: red, name: "c2")
rect((0, 2), (1, 2.5), name: "r1")
rect-around("c1", "c2", "r1", stroke: yellow, padding: 0.1)
```

Styling

The padding attribute can be used to control spacing. Other attributes are forwarded to the rect shape.

Anchors

The same as for the rect shape.

Parameters

- **`..pts-style`** `coordinates` `style`

  Positional two or more coordinates/elements to calculate bounding box of. Accepts style key-value pairs.

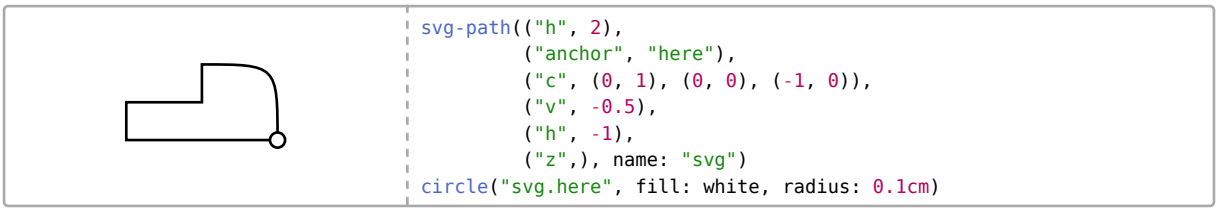
svg-path

```
svg-path(
  name: none string,
  anchor: none coordinate,
  ..commands-style any
)
```

Create a new path from a SVG-like list of commands.

The following commands are supported (uppercase command names use absolute coordinates, lowercase use relative coordinates)

- `("l", coordinate)` line to `coordinate`
- `("h", number)` Horizontal line
- `("v", number)` Vertical line
- `("m", coordinate)` Move to `coordinate`
- `("c", ctrl-coordinate-a, ctrl-coordinate-b, coordinate)` Cubic bezier curve to `coordinate` with two control points a and b
- `("q", ctrl-coordinate, coordinate)` Quadratic bezier curve
- `("z",)` Close the current path
- `("anchor", "<anchor-name>", [coordinate=(0, 0)])` named anchor. If the anchor coordinate is unset, the default `(0, 0)` is used. The anchor named "default" serves as origin for the `anchor:` argument.



```
svg-path(("h", 2),
         ("anchor", "here"),
         ("c", (0, 1), (0, 0), (-1, 0)),
         ("v", -0.5),
         ("h", -1),
         ("z",), name: "svg")
circle("svg.here", fill: white, radius: 0.1cm)
```

Parameters

- **`name`** `none` `string`
- **`anchor`** `none` `coordinate`
- **`..commands-style`** `any`

  Path commands and style keys

4.3 Styling

set-style

```
set-style(
  ..style style
)
```

Set current style

Parameters

- **`..style`** `style`

  Style key-value pairs

fill

```
fill(
  fill paint
)
```

Set current fill style

Shorthand for `set-style(fill: <fill>)`

Parameters

- **`fill`** `paint`

  Fill style

stroke

```
stroke(
  stroke stroke
)
```

Set current stroke style

Shorthand for `set-style(stroke: <fill>)`

Parameters

- **`stroke`** `stroke`

  Stroke style

register-mark

```
register-mark(
  symbol str,
  body function,
  mnemonic: none str,
  tip: none number coordinate,
  base: none number coordinate,
  center: none number coordinate,
  reverse-tip: none number coordinate,
  reverse-base: none number coordinate,
  reverse-center: none number coordinate
)
```

Register a custom mark to the canvas

The mark should contain both anchors called **tip** and **base** that are used to determine the marks orientation. If unset both default to `(0, 0)`. An anchor named **center** is used as center of the mark, if present. Otherwise the mid between **tip** and **base** is used.

```
register-mark(":)", style => {
  circle((0,0), radius: .5, fill: yellow)
  arc((0,0), start: 180deg + 30deg, delta: 180deg - 60deg, anchor:
"origin", radius: .3)
  circle((-0.15, 0.15), radius: .1, fill: white)
  circle((-0.10, 0.10), radius: .025, fill: black)
  circle(( 0.15, 0.15), radius: .1, fill: white)
  circle(( 0.20, 0.10), radius: .025, fill: black)

  anchor("tip",  ( 0.5, 0))
  anchor("base", (-0.5, 0))
})

line((0,0), (3,0), mark: (end: ":)"))
```

Parameters

- **`symbol`** `str`
  Mark name
- **`mnemonic`** `none` `str`
  Mark short name
- **`body`** `function`
  Mark drawing callback, receiving the mark style as argument and returning elements. Format `(styles) => elements`.
- **`tip`** `none` `number` `coordinate`
  Tip coordinate (if passed a number, the y component is 0)
- **`base`** `none` `number` `coordinate`
  Base coordinate (see tip)
- **`center`** `none` `number` `coordinate`
  Center coordinate (see tip)
- **`reverse-tip`** `none` `number` `coordinate`
  Reversed tip coordinate (see tip)
- **`reverse-base`** `none` `number` `coordinate`
  Reversed base coordinate (see tip)
- **`reverse-center`** `none` `number` `coordinate`
  Reversed center coordinate (see tip)

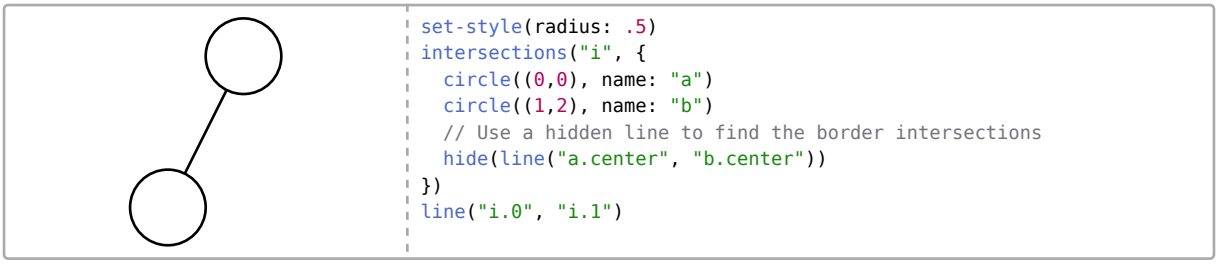
4.4 Grouping

hide

```
hide(
  body element,
  bounds: bool
)
```

Hides an element.

Hidden elements are not drawn to the canvas, are ignored when calculating bounding boxes and discarded by [merge-path](../shapes/merge-path). All other behaviours remain the same as a non-hidden element.



```
set-style(radius: .5)
intersections("i", {
  circle((0,0), name: "a")
  circle((1,2), name: "b")
  // Use a hidden line to find the border intersections
  hide(line("a.center", "b.center"))
})
line("i.0", "i.1")
```

Parameters

- **body** `element`

  One or more elements to hide

- **bounds** `bool`

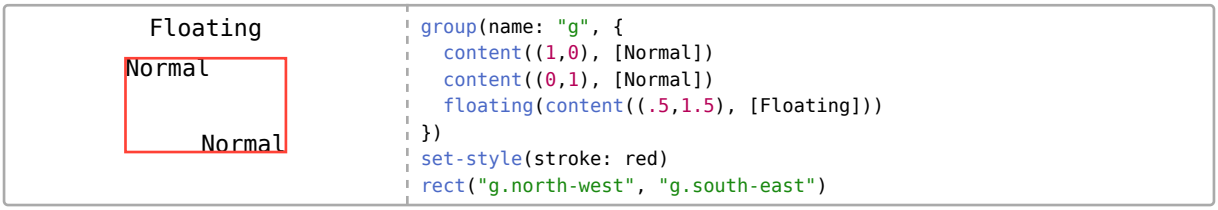
  If true, respect the bounding box of the hidden elements for resizing the canvas

floating

```
floating(
  body element
)
```

Places an element without affecting bounding boxes.

Floating elements are drawn to the canvas but are ignored when calculating bouding boxes. All other behaviours remain the same.




```
group(name: "g", {
  content((1,0), [Normal])
  content((0,1), [Normal])
  floating(content((.5,1.5), [Floating]))
})
set-style(stroke: red)
rect("g.north-west", "g.south-east")
```

Parameters

- **body** `element`

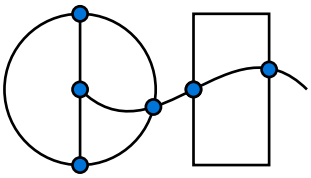
  One or more elements to place

intersections

```
intersections(
  name str,
  ..elements elements str,
  samples: int,
  sort: none function,
  ignore-marks: bool
)
```

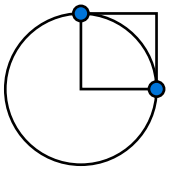
Calculates the intersections between multiple paths and creates one anchor per intersection point.

All resulting anchors will be named numerically, starting at `0`. i.e., a call `intersections("a", ...)` will generate the anchors `"a.0"`, `"a.1"`, `"a.2"` to `"a.n"`, depending of the number of intersections.



```
intersections("i", {
  circle((0, 0))
  bezier((0,0), (3,0), (1,-1), (2,1))
  line((0,-1), (0,1))
  rect((1.5,-1),(2.5,1))
})
for-each-anchor("i", (name) => {
  circle("i." + name, radius: .1, fill: blue)
})
```

You can also use named elements:



```
circle((0,0), name: "a")
rect((0,0), (1,1), name: "b")
intersections("i", "a", "b")
for-each-anchor("i", (name) => {
  circle("i." + name, radius: .1, fill: blue)
})
```

You can calculate intersections with hidden elements by using [hide](./hide).

CeTZ provides the following sorting functions:
- sorting.points-by-distace(points, reference: (0, 0, 0))
- sorting.points-by-angle(points, reference: (0, 0, 0))

Parameters

- **`name`** `str`

  Name to prepend to the generated anchors. (Not to be confused with other `name` arguments that allow the use of anchor coordinates.)

- **`..elements`** `elements` `str`

  Elements and/or element names to calculate intersections with. Elements referred to by name are (unlike elements passed) not drawn by the intersections function!

- **`samples`** `int`

  Number of samples to use for non-linear path segments. A higher sample count can give more precise results but worse performance.

- **`sort`** `none` `function`

  A function of the form `(context, array<vector>) -> array<vector>` that gets called with the list of intersection points.

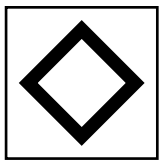
- **`ignore-marks`** `bool`

  If true, ignore mark shapes.

group

```
group(
  body elements function,
  name: none str,
  anchor: none str,
  ..style style
)
```

Groups one or more elements together. This element acts as a scope, all state changes such as transformations and styling only affect the elements in the group. Elements after the group are not affected by the changes inside the group.



```
// Create group
group({
  stroke(5pt)
  scale(.5); rotate(45deg)
  rect((-1,-1),(1,1))
})
rect((-1,-1),(1,1))
```

Styling

Root: `group`

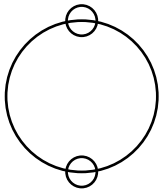
Anchors

Supports border and path anchors of the axis aligned bounding box of all the child elements of the group.

You can add custom named anchors to the group by using the [anchor](./anchor) element while in the scope of said group, see [anchor](./anchor) for more details.

The default anchor is `"center"` but this can be overridden by using [anchor](./anchor) to place a new anchor called `"default"`.

When using named elements within a group, you can access the element's anchors outside of the group by using the implicit anchor coordinate. e.g. `"a.b.north"`



```
group(name: "a", {
  circle((), name: "b")
})
circle("a.b.south", radius: 0.2)
circle((name: "a", anchor: "b.north"), radius: 0.2)
```

Parameters

- **`body`** `elements` `function`

  Elements to group together. A least one is required. A function that accepts `ctx` and returns elements is also accepted.

- **`anchor`** `none` `str`

  Anchor to position the group and it's children relative to. For translation the difference between the groups `"default"` anchor and the passed anchor is used.

- **`name`** `none` `str`

- **`..style`** `style`

- **`padding`** `none` `number` `array` `dictionary`

  How much padding to add around the group's bounding box. `none` applies no padding. A number applies padding to all sides equally. A dictionary applies padding following Typst's `pad` function: https://typst.app/docs/reference/layout/pad/. An array follows CSS like padding: `(y, x)`, `(top, x, bottom)` or `(top, right, bottom, left)`.

scope

```
scope(
  body elements function
)
```

This element acts as a scope, all state changes such as transformations and styling only affect the elements in the scope. Elements after the scope are not affected by the changes inside the scope. In

contrast to `group`, the `scope` element does not create a named element itself and "leaks" body elements and anchors to the outside.

Parameters

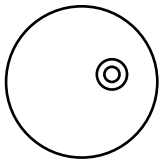
- **`body`** `elements` `function`

  Elements to group together. A least one is required. A function that accepts `ctx` and returns elements is also accepted.

anchor

```
anchor(
  name str,
  position coordinate
)
```

Creates a new anchor for the current group. The new anchor will be accessible from inside the group by using just the anchor's name as a coordinate.



```
// Inside a group
group(name: "g", {
  circle((0,0))
  anchor("x", (.4, .1))
  circle("x", radius: .2)
})
circle("g.x", radius: .1)
```



```
// At the root scope
anchor("x", (1, 1))
// ...
circle("x", radius: .1)
```

Parameters

- **`name`** `str`

  The name of the anchor

- **`position`** `coordinate`

  The position of the anchor

copy-anchors

```
copy-anchors(
  element str,
  filter: auto array
)
```

Copies multiple anchors from one element into the current group. Panics when used outside of a group. Copied anchors will be accessible in the same way anchors created by the `anchor` element are.

Parameters

- **`element`** `str`

  The name of the element to copy anchors from.

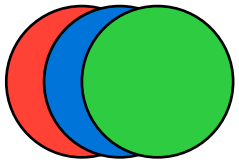
- **`filter`** `auto` `array`

  When set to `auto` all anchors will be copied to the group. An array of anchor names can instead be given so only the anchors that are in the element and the list will be copied over.

set-ctx

```
set-ctx(
  callback function
)
```

An advanced element that allows you to modify the current canvas {{context}}. Note: The transformation matrix (`transform`) is rounded after calling the `callback` function and therefore might be not exactly the matrix specified. This is due to rounding errors and should not cause any problems.



```
// Setting a custom transformation matrix
set-ctx(ctx => {
  let mat = ((1, 0, .5, 0),
             (0, 1,  0, 0),
             (0, 0,  1, 0),
             (0, 0,  0, 1))
  ctx.transform = mat
  return ctx
})
circle((z: 0), fill: red)
circle((z: 1), fill: blue)
circle((z: 2), fill: green)
```

You can store shared context data under a key in the `ctx.shared-data` dictionary. The `ctx.shared-data` dictionary is not scoped by `group` or `scope` elements and can be used for canvas global state.

Parameters

- **`callback`** `function`

  A function that accepts the context dictionary and only returns a new one.

get-ctx

```
get-ctx(
  callback function
)
```

An advanced element that allows you to read the current {{context}} through a callback and return {{element}}s based on it.

```
(
  (1.0, 0.0, 0.0, 0.0),
  (0.0, 1.0, 0.0, 0.0),
  (0.0, 0.0, 1.0, 0.0),
  (0.0, 0.0, 0.0, 1.0),
)
```

```
// Print the transformation matrix
get-ctx(ctx => {
  content((), [#repr(ctx.transform)])
})
```

Parameters

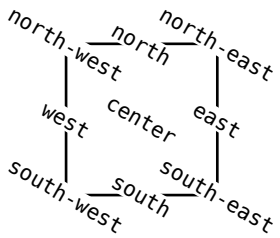
- **`callback`** `function`

  A function that accepts the {{context}} and can return elements.

for-each-anchor

```
for-each-anchor(
  name str,
  callback function,
  exclude: array
)
```

Iterates through all named anchors of an element and calls a callback for each one.




```
// Label nodes anchors
rect((0, 0), (2,2), name: "my-rect")
for-each-anchor("my-rect", exclude: ("start", "mid", "end"), (name)
=> {
   content((), box(inset: 1pt, fill: white, text(8pt, [#name])),
angle: -30deg)
})
```

Parameters

- **name** `str`

  The name of the element with the anchors to loop through.

- **callback** `function`

  A function that takes the anchor name and can return elements.

- **exclude** `array`

  An array of anchor names to not include in the loop.

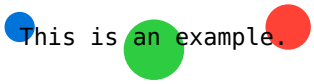
on-layer

```
on-layer(
  layer float int,
  body elements function
)
```

Places elements on a specific layer.

A layer determines the position of an element in the draw queue. A lower layer is drawn before a higher layer.

Layers can be used to draw behind or in front of other elements, even if the other elements were created before or after. An example would be drawing a background behind a text, but using the text's calculated bounding box for positioning the background.




```
// Draw something behind text
set-style(stroke: none)
content((0, 0), [This is an example.], name: "text")
on-layer(-1, {
  circle("text.north-east", radius: .3, fill: red)
  circle("text.south", radius: .4, fill: green)
  circle("text.north-west", radius: .2, fill: blue)
})
```

Parameters

- **layer** `float` `int`

  The layer to place the elements on. Elements placed without `on-layer` are always placed on layer 0.

- **body** `elements` `function`

  Elements to draw on the layer specified. A function that accepts `ctx` and returns elements is also accepted.

4.5 Transformation

set-transform

```
set-transform(
  mat none matrix
)
```

Sets the transformation matrix.

Parameters

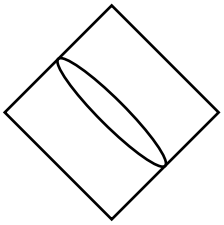
- **mat** none matrix

  The 4x4 transformation matrix to set. If none is passed, the transformation matrix is set to the identity matrix (matrix.ident()).

rotate

```
rotate(
  ..angles angle,
  origin: none coordinate
)
```

Rotates the transformation matrix on the z-axis by a given angle or other axes when specified.



```
// Rotate on z-axis
rotate(z: 45deg)
rect((-1,-1), (1,1))
// Rotate on y-axis
rotate(y: 80deg)
circle((0,0))
```

Parameters

- **..angles** angle

  A single angle as a positional argument to rotate on the z-axis by. Named arguments of x, y or z can be given to rotate on their respective axis. You can give named arguments of yaw, pitch or roll, too.

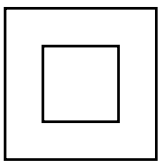
- **origin** none coordinate

  Origin to rotate around, or (0, 0, 0) if set to none.

translate

```
translate(
  ..args vector float length,
  pre: bool
)
```

Translates the transformation matrix by the given vector or dictionary.



```
// Outer rect
rect((0, 0), (2, 2))
// Inner rect
translate(x: .5, y: .5)
rect((0, 0), (1, 1))
```

Parameters

- **`..args`** `vector` `float` `length`

  A single vector or any combination of the named arguments `x`, `y` and `z` to translate by. A translation matrix with the given offsets gets multiplied with the current transformation depending on the value of `pre`.

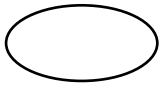
- **`pre`** `bool`

  Specify matrix multiplication order - false: `World = World * Translate` - true: `World = Translate * World`

scale

```
scale(
  ..args float ratio,
  origin: none coordinate
)
```

Scales the transformation matrix by the given factor(s).



```
// Scale the y-axis
scale(y: 50%)
circle((0,0))
```

Note that content like text does not scale automatically. See `auto-scale` styling of content for that.

Parameters

- **`..args`** `float` `ratio`

  A single value to scale the transformation matrix by or per axis scaling factors. Accepts a single float or ratio value or any combination of the named arguments `x`, `y` and `z` to set per axis scaling factors. A ratio of 100% is the same as the value 1.

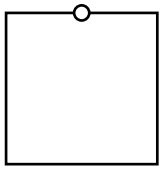
- **`origin`** `none` `coordinate`

  Origin to rotate around, or (0, 0, 0) if set to `none`.

set-origin

```
set-origin(
  origin coordinate
)
```

Sets the given position as the new origin `(0, 0, 0)`



```
// Draw some rect
rect((0,0), (2,2), name: "r")

// Move (0, 0) to the top edge of “r”
set-origin("r.north")
circle((0, 0), radius: .1, fill: white)
```

Parameters

- **`origin`** `coordinate`

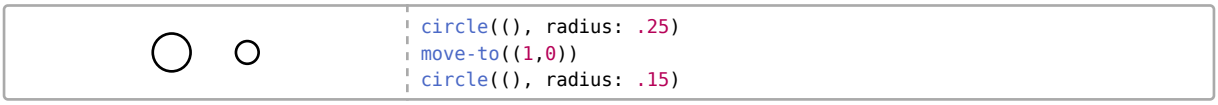
  Coordinate to set as new origin `(0,0,0)`

move-to

```
move-to(
  pt coordinate
)
```

Sets the previous coordinate.

The previous coordinate can be used via `()` (empty coordinate). It is also used as base for relative coordinates if not specified otherwise.



```
circle((), radius: .25)
move-to((1,0))
circle((), radius: .15)
```

Parameters

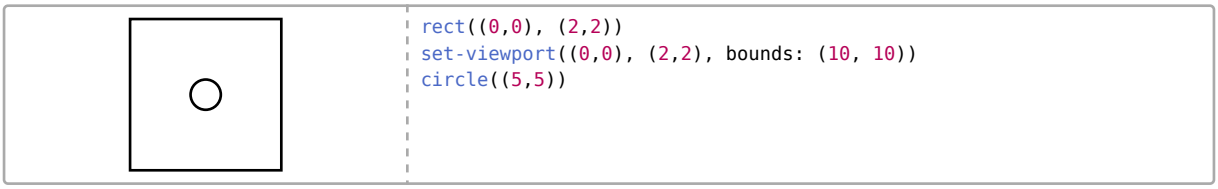
- **pt** `coordinate`

  The coordinate to move to.

set-viewport

```
set-viewport(
  from coordinate,
  to coordinate,
  bounds: vector
)
```

Span viewport between two coordinates and set-up scaling and translation



```
rect((0,0), (2,2))
set-viewport((0,0), (2,2), bounds: (10, 10))
circle((5,5))
```

Parameters

- **from** `coordinate`

  Bottom left corner coordinate

- **to** `coordinate`

  Top right corner coordinate

- **bounds** `vector`

  Viewport bounds vector that describes the inner width, height and depth of the viewport

4.6 Projection

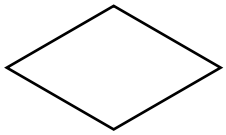
ortho

```
ortho(
  x: angle,
  y: angle,
  z: angle,
  sorted: bool,
  cull-face: none str,
  reset-transform: bool,
  body element
)
```

Set-up an orthographic projection environment.

This is a transformation matrix that rotates elements around the x, the y and the z axis by the parameters given.

By default an isometric projection (x ≈ 35.264°, y = 45°) is set.



```
ortho({
  on-xz({
    rect((-1,-1), (1,1))
  })
})
```

Parameters

- **x** `angle`

  X-axis rotation angle
- **y** `angle`

  Y-axis rotation angle
- **z** `angle`

  Z-axis rotation angle
- **sorted** `bool`

  Sort drawables by maximum distance (front to back)
- **cull-face** `none` `str`

  Enable back-face culling if set to `"cw"` for clockwise or `"ccw"` for counter-clockwise. Polygons of the specified order will not get drawn.
- **reset-transform** `bool`

  Ignore the current transformation matrix
- **body** `element`

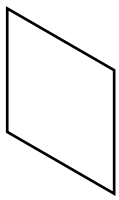
  Elements to draw

on-xy

```
on-xy(
  z: number,
  body element
)
```

Draw elements on the xy-plane with optional z offset.

All vertices of all elements will be changed in the following way: $(x \ \ y \ \ z_{\text{argument}})$, where z_{argument} is the z-value given as argument.



```
ortho({
  on-xy({
    rect((-1, -1), (1, 1))
  })
})
```

Parameters

- **z** `number`

  Z offset for all coordinates
- **body** `element`

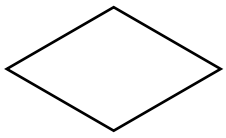
  Elements to draw

on-xz

```
on-xz(
  y: number,
  body element
)
```

Draw elements on the xz-plane with optional y offset.

All vertices of all elements will be changed in the following way: $(x \;\; y_{\text{argument}} \;\; y)$, where y_{argument} is the y-value given as argument.



```
ortho({
  on-xz({
    rect((-1, -1), (1, 1))
  })
})
```

Parameters

- **y** `number`

  Y offset for all coordinates

- **body** `element`

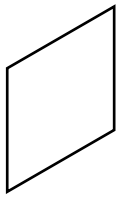
  Elements to draw

on-yz

```
on-yz(
  x: number,
  body element
)
```

Draw elements on the yz-plane with optional x offset.

All vertices of all elements will be changed in the following way: $(x_{\text{argument}} \;\; y \;\; x)$, where x_{argument} is the x-value given as argument.



```
ortho({
  on-yz({
    rect((-1, -1), (1, 1))
  })
})
```

Parameters

- **x** `number`

  X offset for all coordinates

- **body** `element`

  Elements to draw

4.7 Utility

assert-version

```
assert-version(
  min version,
  max: none version,
```

```
  hint: string
)
```

Assert that the cetz version of the canvas matches the given version (range).

Parameters

- **`min`** `version`
  Minimum version (current >= min)
- **`max`** `none` `version`
  First unsupported version (current < max)
- **`hint`** `string`
  Name of the function/module this assert is called from

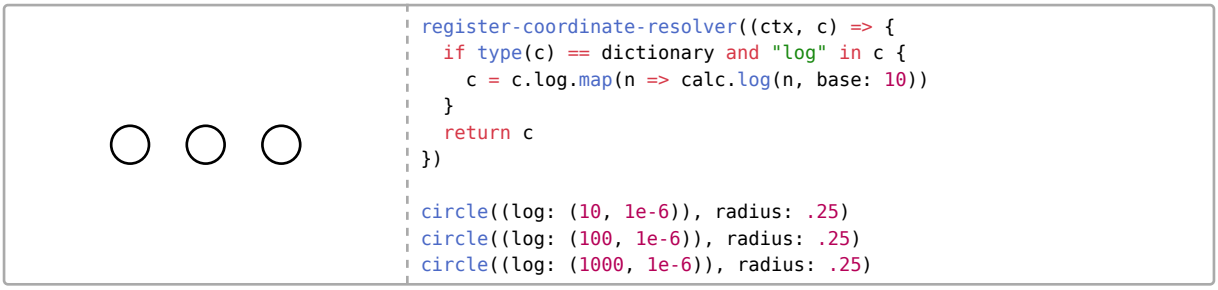
register-coordinate-resolver

```
register-coordinate-resolver(
  resolver function
)
```

Push a custom coordinate resolve function to the list of coordinate resolvers. This resolver is scoped to the current context scope!

A coordinate resolver must be a function of the format `(context, coordinate) => coordinate`. And must *always* return a valid coordinate or panic, in case of an error.

If multiple resolvers are registered, coordinates get passed through all resolvers in reverse registering order. All coordinates get paased to cetz' default coordinate resolvers.



```
register-coordinate-resolver((ctx, c) => {
  if type(c) == dictionary and "log" in c {
    c = c.log.map(n => calc.log(n, base: 10))
  }
  return c
})

circle((log: (10, 1e-6)), radius: .25)
circle((log: (100, 1e-6)), radius: .25)
circle((log: (1000, 1e-6)), radius: .25)
```

Parameters

- **`resolver`** `function`
  The resolver function, taking a context and a single coordinate and returning a single coordinate

4.8 Libraries

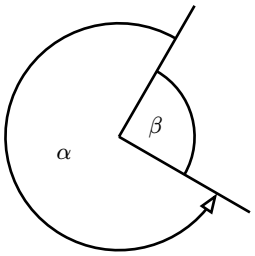
Angle

angle

```
angle(
  origin coordinate,
  a coordinate,
  b coordinate,
  direction: string,
  label: none content function,
  name: none str,
```

  ..style style
)

Draw an angle counter-clock-wise between a and b through origin origin




```
line((0, 0), (60deg, 2), name: "a")
line((0, 0), (330deg, 2), name: "b")

// Draw an angle between the two lines
cetz.angle.angle("a.start", "a.end", "b.end", label: $alpha$,
  mark: (end: ">"), radius: 1.5)
cetz.angle.angle("a.start", "b.end", "a.end", label: $beta$,
  radius: 50%, direction: "ccw")
```

Styling
Root: angle

Anchors
a Point a
b Point b
origin Origin
label Label center
start Arc start
end Arc end

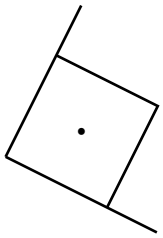
Parameters

- **origin** coordinate
  Angle origin
- **a** coordinate
  Coordinate of side a, containing an angle between origin and b.
- **b** coordinate
  Coordinate of side b, containing an angle between origin and a.
- **direction** string
  Direction of the angle. Accepts “ccw” (counter-clockwise), “cw” (clockwise), “near” (inner angle), “far” (outer angle), the first one being the default.
- **label** none content function
  Draw a label at the angles “label” anchor. If label is a function, it gets the angle value passed as argument. The function must be of the format angle => content.
- **name** none str
  Element name, used for querying anchors.
- **..style** style
  Style key-value pairs.
- **radius** number
  The radius of the angles arc. If of type ratio, it is relative to the smaller distance of either origin to a or origin to b.
- **label-radius** number ratio
  The radius of the angles label origin. If of type ratio, it is relative to radius.

right-angle

```
right-angle(
  origin coordinate,
  a coordinate,
  b coordinate,
  label: none content,
  name: none str,
  ..style style
)
```

Draw a right angle between `a` and `b` through origin `origin`



```
line((0,0), (1,2), name: "a")
line((0,0), (2,-1), name: "b")

// Draw an angle between the two lines
cetz.angle.right-angle(
  "a.start",
  "a.end",
  "b.end",
  radius: 1.5
)
```

Styling

Styling is the same as the `angle` function.

Anchors

Anchors are the same as the `angle` function

Parameters

- **`origin`** `coordinate`

  Angle origin
- **`a`** `coordinate`

  Coordinate of side `a`, containing an angle between `origin` and `b`.
- **`b`** `coordinate`

  Coordinate of side `b`, containing an angle between `origin` and `a`.
- **`label`** `none` `content`

  Draw a label at the angles "label" anchor.
- **`name`** `none` `str`

  Element name, used for querying anchors.
- **`..style`** `style`

  Style key-value pairs.

Tree

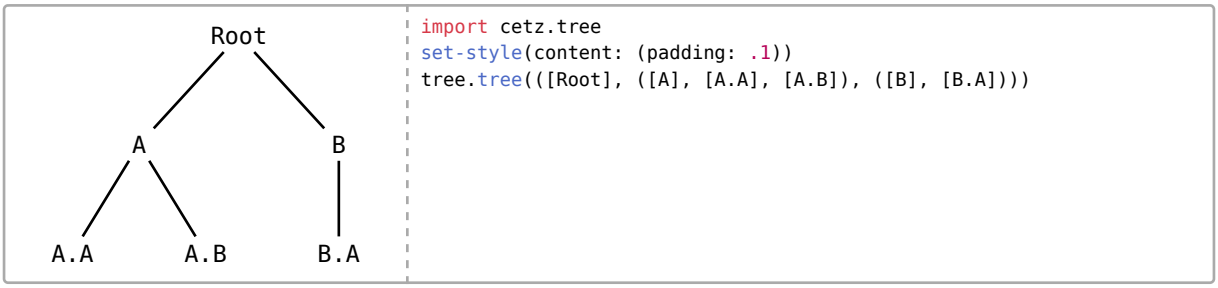
tree

```
tree(
  root array,
  draw-node: auto function,
  draw-edge: none auto function,
  direction: str,
  grow: float,
```

```
  spread: float,
  name: none str,
  node-layer: int,
  edge-layer: int,
  measure-content: ,
  anchor: none string
)
```

Lays out and renders tree nodes.

For each node, the `tree` function creates an anchor of the format `"[<child-index>-]<child-index>"` (the root is `"0"`, its first child `"0-0"`, second `"0-1"` and so on) that can be used to query a nodes position on the canvas.




```
import cetz.tree
set-style(content: (padding: .1))
tree.tree(([Root], ([A], [A.A], [A.B]), ([B], [B.A])))
```

Parameters

- **root** `array`

  A nested array of content that describes the structure the tree should take. Example: `([root], [child 1], ([child 2], [grandchild 1]))`

- **draw-node** `auto` `function`

  The function to call to draw a node. The function will be passed the node to draw (a dictionary with a `content` key) and is expected to return elements (`(node, parent-node) => elements`). The node must be drawn at the `(0,0)` coordinate. If `auto` is given, just the node's value will be drawn as content.

- **draw-edge** `none` `auto` `function`

  The function to call draw an edge between two nodes. The function will be passed the name of the starting node, the name of the ending node, the start node, the end node, and is expected to return elements (`(source-name, target-name, parent-node, child-node) => elements`). If `auto` is given, a straight line will be drawn between nodes.

- **direction** `str`

  A string describing the direction the tree should grow in ("up", "down", "left", "right")

- **grow** `float`

  Depth grow factor

- **spread** `float`

  Sibling spread factor

- **name** `none` `str`

  The tree element's name

- **node-layer** `int`

  Layer to draw nodes on

- **edge-layer** `int`

  Layer to draw edges on

- **`anchor`** `none` `string`
  Name of the anchor to align the tree to. Use the root node anchor (`"0"`) to align the tree to the root nodes position.

Decorations

Path

zigzag

```
zigzag(
  target drawable,
  name: none string,
  close: auto bool,
  ..style style
)
```

Draw a zig-zag or saw-tooth wave along a path.

The number of tooths can be controlled via the `segments` or `segment-length` style key, and the width via `amplitude`.

```
line((0,0), (2,1), stroke: gray)
cetz.decorations.zigzag(line((0,0), (2,1)), amplitude: .25, start:
10%, stop: 90%)
```

Styling

Root: `zigzag`

Parameters

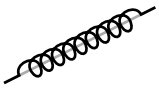
- **`target`** `drawable`
  Target path
- **`close`** `auto` `bool`
  Close the path
- **`name`** `none` `string`
  Element name
- **`..style`** `style`
  Style
- **`factor`** `ratio`
  Triangle mid between its start and end. Setting this to 0% leads to a falling sawtooth shape, while 100% results in a raising sawtooth.

coil

```
coil(
  target drawable,
  close: auto bool,
  name: none string,
  ..style style
)
```

Draw a stretched coil/loop spring along a path

The number of windings can be controlled via the `segments` or `segment-length` style key, and the width via `amplitude`.



```
line((0,0), (2,1), stroke: gray)
cetz.decorations.coil(line((0,0), (2,1)), amplitude: .25, start:
10%, stop: 90%)
```

Styling
Root: `coil`

Parameters

- **`target`** `drawable`
  Target path
- **`close`** `auto` `bool`
  Close the path
- **`name`** `none` `string`
  Element name
- **`..style`** `style`
  Style
- **`factor`** `ratio`
  Factor of how much the coil overextends its length to form a curl.

wave

```
wave(
  target drawable,
  close: auto bool,
  name: none string,
  ..style style
)
```

Draw a wave along a path using a catmull-rom curve

The number of phases can be controlled via the `segments` or `segment-length` style key, and the width via `amplitude`.

```
line((0,0), (2,1), stroke: gray)
cetz.decorations.wave(line((0,0), (2,1)), amplitude: .25, start:
10%, stop: 90%)
```

Styling
Root: `wave`

- tension (float) = 0.5 Catmull-Rom curve tension, see [Catmull](/api/draw-functions/shapes/catmull)

Parameters

- **`target`** `drawable`
  Target path
- **`close`** `auto` `bool`
  Close the path
- **`name`** `none` `string`
  Element name

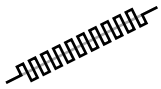
- **`..style`** `style`
  Style

square

```
square(
  target drawable,
  close: auto bool,
  name: none string,
  ..style style
)
```

Draw a square-wave along a path using a line-strip

The number of phases can be controlled via the `segments` or `segment-length` style key, and the width via `amplitude`.



```
line((0,0), (2,1), stroke: gray)
cetz.decorations.square(line((0,0), (2,1)), amplitude: .25, start:
10%, stop: 90%)
```

Styling

Root: `squre`

- factor (ratio) = 50% Square-Wave midpoint

Parameters

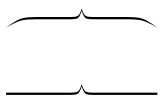
- **`target`** `drawable`
  Target path
- **`close`** `auto` `bool`
  Close the path
- **`name`** `none` `string`
  Element name
- **`..style`** `style`
  Style

Brace

brace

```
brace(
  start coordinate,
  end coordinate,
  ..style style,
  name: string none
)
```

Draw a curly brace between two points.



```
cetz.decorations.brace((0,1),(2,1))
cetz.decorations.brace((0,0),(2,0), outer-inset: 0)
```

Styling

Root: `brace`

Use the `fill` style for tapered braces and set `stroke` to none.

Anchors

start Where the brace starts, same as the `start` parameter.
end Where the brace end, same as the `end` parameter.
spike Point of the spike, halfway between `start` and `end` and shifted by `amplitude` towards the pointing direction.
content Point to place content/text at, in front of the spike.
center Center of the enclosing rectangle.

Parameters

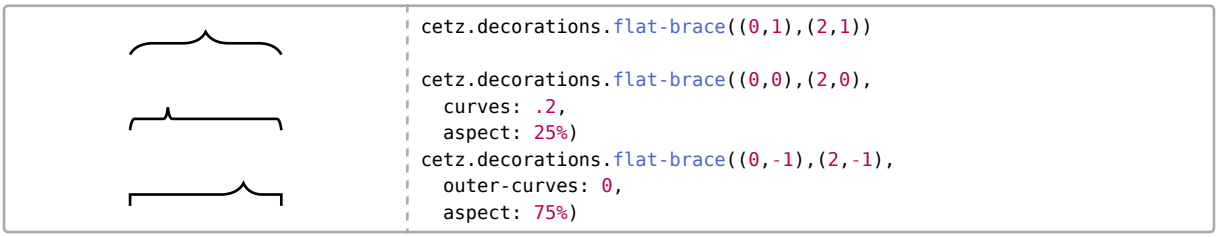
- **`start`** `coordinate`
  Start point
- **`end`** `coordinate`
  End point
- **`name`** `string` `none`
  Element name used for querying anchors
- **`..style`** `style`
  Style key-value pairs
- **`amplitude`** `number`
  Sets the height of the brace, from its baseline to its middle tip.
- **`thickness`** `number` `ratio`
  Thickness of tapered braces (if ratio, relative to half the amplitude).
- **`pointiness`** `ratio`
  Thickness of the mid-spice
- **`taper`** `bool`
  Draw a tapered brace
- **`outer-inset`** `number` `ratio`
  Inset of the outer curve points
- **`outer-curvyness`** `ratio`
  Curvyness of the outer curves
- **`inner-outset`** `number` `ratio`
  Inset of the inner tip curve points
- **`inner-curvyness`** `ratio`
  Curvyness of the inner tip curves
- **`outer-thickness`** `number`
  Thickness of the outer tips
- **`content-offset`** `number`
  Offset of the `"content"` anchor from the spike of the brace.
- **`flip`** `bool`
  Mirror the brace along the line between start and end.

flat-brace

```
flat-brace(
  start coordinate,
  end coordinate,
```

```
  flip: bool,
  debug: bool,
  name: str none,
  ..style style
)
```

Draw a flat curly brace between two points.



```
cetz.decorations.flat-brace((0,1),(2,1))

cetz.decorations.flat-brace((0,0),(2,0),
  curves: .2,
  aspect: 25%)
cetz.decorations.flat-brace((0,-1),(2,-1),
  outer-curves: 0,
  aspect: 75%)
```

This mimics the braces from TikZ's [`decorations.pathreplacing` library](https://github.com/pgf-tikz/pgf/blob/6e5fd71581ab04351a89553a259b57988bc28140/tex/generic/pgf/libraries/decorations/pgflibrarydecorations.pathreplacing.code.tex#L136-L185). In contrast to the `brace` function, these braces use straight line segments, resulting in better looks for long braces with a small amplitude.

Styling

Root: `flat-brace`

- aspect (ratio) = 50% Determines the fraction of the total length where the spike will be placed.

Anchors

start Where the brace starts, same as the `start` parameter.
end Where the brace end, same as the `end` parameter.
spike Point of the spike's top.
content Point to place content/text at, in front of the spike.
center Center of the enclosing rectangle.

Parameters

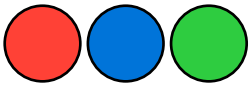
- **`start`** `coordinate`
  Start point
- **`end`** `coordinate`
  End point
- **`flip`** `bool`
  Flip the brace around
- **`name`** `str` `none`
  Element name for querying anchors
- **`debug`** `bool`
- **`..style`** `style`
  Style key-value pairs
- **`amplitude`** `number`
  Determines how much the brace rises above the base line.
- **`curves`** `number` `auto` `array`
  Curviness factor of the brace, a factor of 0 means no curves.
- **`outer-curves`** `number` `auto` `array`
  Curviness factor of the outer curves of the brace. A factor of 0 means no curves.

Palette

new

```
new(
  base: style,
  colors: none array,
  dash: none array
) ⟶ function
```

Create a new palette based on a base style



```
let p = cetz.palette.new(colors: (red, blue, green))
for i in range(0, p("len")) {
  set-style(..p(i))
  circle((0,0), radius: .5)
  set-origin((1.1, 0))
}
```

The functions returned by this function have the following named arguments:

You can use a palette for stroking via: `red.with(stroke: true)`.

Parameters

- **`fill`** `bool`
  If true, the returned fill color is one of the colors from the `colors` list, otherwise the base styles fill is used.
- **`stroke`** `bool`
  If true, the returned stroke color is one of the colors from the `colors` list, otherwise the base styles stroke color is used.
- **`base`** `style`
  Style dictionary to use as base style for the styles generated per color
- **`colors`** `none` `array`
  List of colors the returned palette should return styles with.
- **`dash`** `none` `array`
  List of stroke dash patterns the returned palette should return styles with.

4.9 Internals

Complex

re

```
re(
  V complex
) ⟶ float
```

Returns the real part of a complex number.

Parameters

- **`V`** `complex`
  A complex number.

im

```
im(
  V complex
) ⟶ float
```

Returns the imaginary part of a complex number.

Parameters

- **V** `complex`

  A complex number.

mul

```
mul(
  V complex,
  W complex
)
```

Multiplies two complex numbers together and returns the result VW.

Parameters

- **V** `complex`

  The complex number on the left hand side.
- **W** `complex`

  The complex number on the right hand side.

conj

```
conj(
  V complex
) ⟶ complex
```

Calculates the conjugate of a complex number.

Parameters

- **V** `complex`

  A complex number.

dot

```
dot(
  V complex,
  W complex
) ⟶ float
```

Calculates the dot product of two complex numbers in R^2 $Vc \cdot W$.

Parameters

- **V** `complex`

  The complex number on the left hand side.
- **W** `complex`

  The complex number on the right hand side.

normsq

```
normsq(
  V complex
) ⟶ float
```

Calculates the squared normal of a complex number.

Parameters

- **`V complex`**
  The complex number.

norm

```
norm(
  V complex
) ⟶ float
```

Calculates the normal of a complex number

Parameters

- **`V complex`**
  The complex number.

scale

```
scale(
  V complex,
  t float
) ⟶ complex
```

Multiplies a complex number by a scale factor.

Parameters

- **`V complex`**
  The complex number to scale.
- **`t float`**
  The scale factor.

unit

```
unit(
  V complex
) ⟶ vector
```

Returns a unit vector in the direction of a complex number.

Parameters

- **`V complex`**
  The complex number.

inv

```
inv(
  V complex
) ⟶ complex
```

Inverts a complex number.

Parameters

- **`V`** `complex`
  The complex number

div

```
div(
  V complex,
  W complex
) ⟶ complex
```

Divides two complex numbers.

Parameters

- **`V`** `complex`
  The complex number of the numerator.
- **`W`** `complex`
  The complex number of the denominator.

add

```
add(
  V complex,
  W complex
) ⟶ complex
```

Adds two complex numbers together.

Parameters

- **`V`** `complex`
  The complex number on the left hand side.
- **`W`** `complex`
  The complex number on the right hand side.

sub

```
sub(
  V complex,
  W complex
) ⟶ complex
```

Subtracts two complex numbers together.

Parameters

- **`V`** `complex`
  The complex number on the left hand side.
- **`W`** `complex`
  The complex number on the right hand side.

arg

```
arg(
  V complex
)
```

Calculates the argument of a complex number.

Parameters

- **`V`** `complex`
  The complex number.

ang

```
ang(
  V complex,
  W complex
)
```

Get the signed angle of two complex numbers from V to W.

Parameters

- **`V`** `complex`
  A complex number.
- **`W`** `complex`
  A complex number.

Vector

as-mat

```
as-mat(
  v vector,
  mode: str
) ⟶ matrix
```

Converts a vector to a row or column matrix.

Parameters

- **`v`** `vector`
  The vector to convert.
- **`mode`** `str`
  The type of matrix to convert into. Must be one of `"row"` or `"column"`.

as-vec

```
as-vec(
  v vector,
  init: vector
) ⟶ vector
```

Ensures a vector has an exact number of components. This is done by passing another vector `init` that has the required dimension. If the original vector does not have enough dimensions, the values from `init` will be inserted. It is recommended to use a zero vector for `init`.

Parameters

- **`v`** `vector`
  The vector to ensure.
- **`init`** `vector`
  The vector to check the dimension against.

len

```
len(
  v vector
) ⟶ float
```

Return length/magnitude of a vector.

Parameters

- **v** `vector`

  The vector to find the magnitude of.

add

```
add(
  v1 vector,
  v2 vector
) ⟶ vector
```

Adds two vectors of the same dimension

Parameters

- **v1** `vector`

  The vector on the left hand side.
- **v2** `vector`

  The vector on the right hand side.

sub

```
sub(
  v1 vector,
  v2 vector
) ⟶ vector
```

Subtracts two vectors of the same dimension

Parameters

- **v1** `vector`

  The vector on the left hand side.
- **v2** `vector`

  The vector on the right hand side.

dist

```
dist(
  a vector,
  b vector
) ⟶ float
```

Calculates the distance between two vectors by subtracting the length of vector `a` from vector `b`.

Parameters

- **a** `vector`

  Vector a
- **b** `vector`

  Vector b

scale

```
scale(
  v vector,
  x float
) ⟶ vector
```

Multiplys a vector with scalar x

Parameters

- **v** `vector`
  The vector to scale.
- **x** `float`
  The scale factor.

div

```
div(
  v vector,
  x float
)
```

Divides a vector by scalar x

Parameters

- **v** `vector`
  The vector to be divded.
- **x** `float`
  The inverse scale factor.

neg

```
neg(
  v vector
) ⟶ vector
```

Negates each value in a vector

Parameters

- **v** `vector`
  The vector to negate.

norm

```
norm(
  v vector
) ⟶ vector
```

Normalizes a vector (divide by its length)

Parameters

- **v** `vector`
  The vector to normalize.

element-product

```
element-product(
  a vector,
  b vector
)
```

Multiply two vectors component-wise

Parameters

- **a** `vector`

  First vector.

- **b** `vector`

  Second vector.

dot

```
dot(
  v1 vector,
  v2 vector
) ⟶ float
```

Calculates the dot product between two vectors.

Parameters

- **v1** `vector`

  The vector on the left hand side.

- **v2** `vector`

  The vector on the right hand side.

cross

```
cross(
  v1 vector,
  v2 vector
) ⟶ vector
```

Calculates the cross product of two vectors with a dimension of three.

Parameters

- **v1** `vector`

  The vector on the left hand side.

- **v2** `vector`

  The vector on the right hand side.

angle2

```
angle2(
  a vector,
  b vector
) ⟶ angle
```

Calculates the angle between two vectors and the x-axis in 2d space

Parameters

- **`a`** `vector`
  The vector to measure the angle from.
- **`b`** `vector`
  The vector to measure the angle to.

angle

```
angle(
  v1 vector,
  c vector,
  v2 vector
)
```

Calculates the angle between three vectors

Parameters

- **`v1`** `vector`
  The vector to measure the angle from.
- **`c`** `vector`
  The vector to measure the angle at.
- **`v2`** `vector`
  The vector to measure the angle to.

lerp

```
lerp(
  v1 vector,
  v2 vector,
  t float
)
```

Linear interpolation between two vectors.

Parameters

- **`v1`** `vector`
  The vector to interpolate from.
- **`v2`** `vector`
  The vector to interpolate to.
- **`t`** `float`
  The factor to interpolate by. A value of `0` is `v1` and a value of `1` is `v2`.

Matrix

ident

```
ident(
  size int
) ⟶ matrix
```

Create a (square) identity matrix with dimensions size × size

Parameters

- **`size`** `int`
  Size of the matrix

diag

```
diag(
  ..diag float
) → matrix
```

Create a square matrix with the diagonal set to the given values

Parameters

- **`..diag`** `float`
  Diagonal values

dim

```
dim(
  m matrix
) → array
```

Returns the dimension of the given matrix as `(m, n)`

Parameters

- **`m`** `matrix`
  The matrix

column

```
column(
  mat matrix,
  n int
) → vector
```

Returns the n-th column of a matrix as a {{vector}}

Parameters

- **`mat`** `matrix`
  Input matrix
- **`n`** `int`
  The column's index

round

```
round(
  mat matrix,
  precision: int
) → matrix
```

Rounds each value in the matrix to a precision.

Parameters

- **`mat`** `matrix`
  Input matrix

- **precision** `int`

  Rounding precision (digits)

transform-translate

```
transform-translate(
  x float,
  y float,
  z float
) ⟶ matrix
```

Returns a 4×4 translation matrix

Parameters

- **x** `float`

  The translation in the x direction.
- **y** `float`

  The translation in the y direction.
- **z** `float`

  The translation in the x direction.

transform-shear-x

```
transform-shear-x(
  factor float
) ⟶ matrix
```

Returns a 4×4 x-shear matrix

Parameters

- **factor** `float`

  The shear in the x direction.

transform-shear-z

```
transform-shear-z(
  factor float
) ⟶ matrix
```

Returns a 4×4 z-shear matrix

Parameters

- **factor** `float`

  The shear in the z direction.

transform-scale

```
transform-scale(
  f float array dictionary
) ⟶ matrix
```

Returns a 4×4 scale matrix

Parameters

- **f** `float` `array` `dictionary`

  The scale factor(s) of the matrix. An {{array}} of at least 3 {{float}}s sets the x, y and z scale factors.

A {{dictionary}} sets the scale in the direction of the corresponding x, y and z keys. A single {{float}} sets the scale for all directions.

transform-rotate-dir

```
transform-rotate-dir(
  dir vector,
  up vector
) ⟶ matrix
```

Returns a 4×4 rotation xyz matrix for a direction and up vector

Parameters

- **dir** `vector`
  idk
- **up** `vector`
  idk

transform-rotate-x

```
transform-rotate-x(
  angle angle
) ⟶ matrix
```

Returns a 4×4 x rotation matrix

Parameters

- **angle** `angle`
  The angle to rotate around the x axis

transform-rotate-y

```
transform-rotate-y(
  angle angle
) ⟶ matrix
```

Returns a 4×4 y rotation matrix

Parameters

- **angle** `angle`
  The angle to rotate around the y axis

transform-rotate-z

```
transform-rotate-z(
  angle angle
) ⟶ matrix
```

Returns a 4×4 z rotation matrix

Parameters

- **angle** `angle`
  The angle to rotate around the z axis

transform-rotate-xz

```
transform-rotate-xz(
  x angle,
```

```
  z angle
) ⟶ matrix
```

Returns a 4×4 xz rotation matrix

Parameters

- **x** `angle`
  The angle to rotate around the x axis
- **z** `angle`
  The angle to rotate around the z axis

transform-rotate-ypr

```
transform-rotate-ypr(
  a angle,
  b angle,
  c angle
) ⟶ matrix
```

Returns a 4×4 rotation matrix - yaw-pitch-roll

Parameters

- **a** `angle`
  Yaw
- **b** `angle`
  Pitch
- **c** `angle`
  Roll

transform-rotate-xyz

```
transform-rotate-xyz(
  x angle,
  y angle,
  z angle
) ⟶ matrix
```

Returns a 4×4 rotation matrix - euler angles

Calculates the product of the three rotation matrices $R = R_{z(z)}R_{y(y)}R_{x(x)}$

Parameters

- **x** `angle`
  Rotation about x
- **y** `angle`
  Rotation about y
- **z** `angle`
  Rotation about z

mul-mat

```
mul-mat(
  ..matrices matrix
) ⟶ matrix
```

Multiplies matrices on top of each other.

Parameters

- **`..matrices`** `matrix`
  The matrices to multiply from left to right.

mul4x4-vec3

```
mul4x4-vec3(
  mat matrix,
  vec vector,
  w: float
) ⟶ vector
```

Multiplies a 4×4 matrix with a vector of size 3 or 4. The resulting is three dimensional

Parameters

- **`mat`** `matrix`
  The matrix to multiply
- **`vec`** `vector`
  The vector to multiply
- **`w`** `float`
  The default value for the fourth element of the vector if it is three dimensional.

mul-vec

```
mul-vec(
  mat matrix,
  vec vector
) ⟶ vector
```

Multiplies an $m \times n$ matrix with an mth dimensional vector where $m \leq 4$. Prefer the use of `mul4x4-vec3` when possible as it does not use loops.

Parameters

- **`mat`** `matrix`
  The matrix to multiply
- **`vec`** `vector`
  The vector to multiply

inverse

```
inverse(
  matrix matrix
) ⟶ matrix
```

Calculates the inverse matrix of any size.

Parameters

- **`matrix`** `matrix`
  The matrix to inverse.

Coordinate

resolve-system

```
resolve-system(
  ctx ,
  c coordinate
) ⟶ str
```

Figures out what system a coordinate belongs to and returns the corresponding string.

Parameters

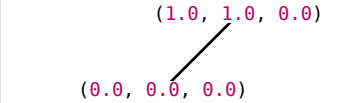
- **c** `coordinate`

  The coordinate to find the system of.

resolve

```
resolve(
  ctx context,
  ..coordinates coordinate,
  update: bool
) ⟶ array
```

Resolve a list of coordinates to absolute vectors. Returns an array of the new context</Type> then the resolved coordinate vectors.




```
line((0,0), (1,1), name: "l")
get-ctx(ctx => {
  // Get the vector of coordinate "l.start" and "l.end"
  let (ctx, a, b) = cetz.coordinate.resolve(ctx, "l.start", "l.end")
  content("l.start", [#a], frame: "rect", stroke: none, fill: white)
  content("l.end",   [#b], frame: "rect", stroke: none, fill: white)
})
```

Parameters

- **ctx** `context`

  Canvas context object

- **..coordinates** `coordinate`

  List of coordinates

- **update** `bool`

  Update the context's last position

Styles

resolve

```
resolve(
  dict style,
  root: none str array,
  merge: style,
  base: none style
) ⟶ style
```

You can use this to combine the style in `ctx`, the style given by a user for a single element and an element's default style.

`base` is first merged onto `dict` without overwriting existing values, and if `root` is given it is merged onto that key of `dict`. `merge` is then merged onto `dict` but does overwrite existing entries, if `root` is given it is merged onto that key of `dict`. Then entries in `dict` that are {{auto}} inherit values from their nearest ancestor and entries of type {{dictionary}} are merged with their closest ancestor.

```
#let dict = (
  stroke: "black",
  fill: none,
  mark: (stroke: auto, fill: "blue"),
  line: (stroke: auto, mark: auto, fill: "red")
)
#cetz.styles.resolve(dict, merge: (mark: (stroke: "yellow")), root: "line")
```

The following is a more detailed explanation of how the algorithm works to use as a reference if needed. It should be updated whenever changes are made. Remember that dictionaries are recursively merged, if an entry is any other type it is simply updated. (dict + dict = merged dict, value + dict = dict, dict + value = value) First if `base` is given, it will be merged without overwriting values onto `dict`. If `root` is given it will be merged onto that key of `dict`. Each level of `dict` is then processed with these steps. If `root` is given the level with that key will be the first, otherwise the whole of `dict` is processed.

1. Values on the corresponding level of `merge` are inserted into the level if the key does not exist on the level or if they are not both dictionaries. If they are both dictionaries their values will be inserted in the same stage at a lower level.
2. If an entry is `auto` or a dictionary, the tree is travelled back up until an entry with the same key is found. If the current entry is `auto` the value of the ancestor's entry is copied. Or if the current entry and ancestor entry is a dictionary, they are merged with the current entry overwriting any values in it's ancestors.
3. Each entry that is a dictionary is then resolved from step 1.

```
(
  scale: 1,
  length: 5.67pt,
  width: 4.25pt,
  inset: 1.42pt,
  sep: 2.83pt,
  pos: none,
  offset: 0,
  start: none,
  end: none,
  symbol: none,
  xy-up: (0, 0, 1),
  z-up: (0, 1, 0),
  stroke: (paint: luma(0%), thickness: 1pt, dash: "solid"),
  fill: none,
  slant: none,
  harpoon: false,
  flip: false,
  reverse: false,
  position-samples: 20,
  shorten-to: auto,
  transform-shape: false,
  anchor: "tip",
)
```

```
get-ctx(ctx => {
  // Get the current "mark" style
  content((0,0), [#cetz.styles.resolve(ctx.style, root: "mark")])
})
```

Parameters

- **`dict`** `style`

  Current context style from `ctx.style`.

- **`merge`** `style`
  Style values overwriting the current style. I.e. inline styles passed with an element: `line(.., stroke: red)`.
- **`root`** `none` `str` `array`
  Style root element name or list of nested roots (`("my-package", "my-element")`).
- **`base`** `none` `style`
  Style values to merge into `dict` without overwriting it.

merge

```
merge(
  bottom ,
  top
)
```

Merge two style dictionaries by using cetz' style folding logic.

- bottom (dictionary) Base style dictionary.
- top (dictionary) New style dictionary to merge on top of `bottom`.

Process

element

```
element(
  ctx ctx,
  element-func function
)
```

Processes an element's function to get its drawables and bounds. Returns a {{dictionary}} with the key-values: `ctx` The modified context object, `bounds` The {{aabb}} of the element's drawables, `drawables` An {{array}} of the element's {{drawable}}s.

Parameters

- **`ctx`** `ctx`
  The current context object.
- **`element-func`** `function`
  A function that when passed {{ctx}}, it should return an element dictionary.

many

```
many(
  ctx ctx,
  body array
) ⟶ dictionary
```

Runs the `element` function for a list of element functions and aggregates the results.

Parameters

- **`ctx`** `ctx`
  The current context object.
- **`body`** `array`
  The array of element functions to process.

Drawable

TAG

```
TAG(

)
```

Tag constants

apply-transform

```
apply-transform(
  transform matrix,
  drawables drawable
) ⟶ drawable
```

Applies a transform to drawables. If a single drawable is given it will be returned in a single element array</Type>.

Parameters

- **`transform`** `matrix`

  The transformation matrix.

- **`drawables`** `drawable`

  The drawables to transform.

apply-tags

```
apply-tags(
  drawables drawable array,
  ..tags str
) ⟶ drawable or array
```

Adds `tags` to one or more drawables.

Parameters

- **`drawables`** `drawable` `array`

  A single drawable or an array of `drawable`.

- **`..tags`** `str`

  The list of tags to add to the drawable

Result

- `drawable`

  A single drawable

- `array`

  An array of `drawable`

filter-tagged

```
filter-tagged(
  drawables drawable array,
  ..tags str
) ⟶ drawable or array
```

Filter out all drawables that have one of the given tags assigned.

Parameters

- **`drawables`** `drawable` `array`

  A single drawable or an array of `drawables`.

- **`..tags`** `str`

  The list of tags to use as a filter.

Result

- `drawable`

  A single drawable

- `array`

  An array of `drawable`

path

```
path(
  fill: color none,
  stroke: stroke,
  fill-rule: str,
  tags: ,
  path
) ⟶ drawable
```

Creates a path drawable from path segements.

Parameters

- **`segments`** `array`

  The segments to create the path from.

- **`close`** `bool`

  If `true` the path will be closed.

- **`fill`** `color` `none`

  The color to fill the path with.

- **`fill-rule`** `str`

  One of “even-odd” or “non-zero”.

- **`stroke`** `stroke`

  The stroke of the path.

line-strip

```
line-strip(
  points array,
  close: bool,
  fill: none fill,
  stroke: none stroke,
  fill-rule: str,
  tags:
) ⟶ drawable
```

Construct a line-strip from a list of points

Parameters

- **`points`** `array`
  Array of points
- **`close`** `bool`
- **`fill`** `none` `fill`
- **`stroke`** `none` `stroke`
- **`fill-rule`** `str`

content

```
content(
  pos vector,
  width float,
  height float,
  border segment,
  body content
) ⟶ drawable
```

Creates a content drawable.

Parameters

- **`pos`** `vector`
  The position of the drawable.
- **`width`** `float`
  The width of the drawable.
- **`height`** `float`
  The height of the drawable.
- **`border`** `segment`
  A segment to define the border of the drawable with.
- **`body`** `content`
  The content of the drawable.

ellipse

```
ellipse(
  x float,
  y float,
  z float,
  rx float,
  ry float,
  fill: color none,
  stroke: stroke
) ⟶ drawable
```

Creates a path drawable in the shape of an ellipse.

Parameters

- **`x`** `float`
  The x position of the ellipse.

- **y** `float`
  The y position of the ellipse.
- **z** `float`
  The z position of the ellipse.
- **rx** `float`
  The radius of the ellipse in the x axis.
- **ry** `float`
  The radius of the ellipse in the y axis.
- **fill** `color` `none`
  The color to fill the ellipse with.
- **stroke** `stroke`
  The stroke of the ellipse's path.

arc

```
arc(
  x float,
  y float,
  z float,
  start angle,
  stop angle,
  rx float,
  ry float,
  mode: str,
  fill: color none,
  stroke: stroke
) ⟶ drawable
```

Creates a path drawable in the shape of an arc.

Parameters

- **x** `float`
  The x position of the start of the arc.
- **y** `float`
  The y position of the start of the arc.
- **z** `float`
  The z position of the start of the arc.
- **start** `angle`
  The angle along an ellipse to start drawing the arc from.
- **stop** `angle`
  The angle along an ellipse to stop drawing the arc at.
- **rx** `float`
  The radius of the arc in the x axis.
- **ry** `float`
  The radius of the arc in the y axis.
- **mode** `str`
  How to draw the arc: `"OPEN"` leaves the path open, `"CLOSED"` closes the arc by drawing a straight

line between the end of the arc and its start, `"PIE"` also closes the arc by drawing a line from its end to its origin then to its start.

- **`fill`** `color` `none`

  The color to fill the arc with.

- **`stroke`** `stroke`

  The stroke of the arc's path.

Anchor

border

```
border(
  center vector,
  x-dist number,
  y-dist number,
  drawables drawables,
  angle angle
) ⟶ none or vector
```

Calculates a border anchor at the given angle by testing for an intersection between a line and the given drawables. Returns none if no intersection is found for better error reporting.

Parameters

- **`center`** `vector`

  The position from which to start the test line.

- **`x-dist`** `number`

  The furthest distance the test line should go in the x direction.

- **`y-dist`** `number`

  The furthest distance the test line should go in the y direction.

- **`drawables`** `drawables`

  Drawables to test for an intersection against. Ideally should be of type path but all others are ignored.

- **`angle`** `angle`

  The angle to check for a border anchor at.

setup

```
setup(
  callback function auto,
  anchor-names array,
  default: str none,
  transform: matrix none,
  name: str none,
  offset-anchor: str none,
  border-anchors: bool,
  path-anchors: bool,
  radii: none array,
  path: none drawable,
  nested-anchors:
) ⟶ array
```

Setup an anchor calculation and handling function for an element. Unifies anchor error checking and calculation of the offset transform.

A tuple of a transformation matrix and function will be returned. The transform is calculated by translating the given transform by the distance between the position of `offset-anchor` and `default`. It can then be used to correctly transform an element's drawables. If either are none the calculation won't happen but the transform will still be returned. The function can be used to get the transformed anchors of an element by passing it a string. An empty array can be passed to get the list of valid anchors.

Parameters

- **`callback`** `function` `auto`
  The function to call to get a named anchor's position. The anchor's name will be passed and it should return a vector</Type> (`str => vector`). If no named anchors exist on the element `auto` can be given instead of a function.
- **`anchor-names`** `array`
  A list of valid anchor names. This list will be used to validate an anchor exists before `callback` is used.
- **`default`** `str` `none`
  The name of the default anchor, if one exists.
- **`transform`** `matrix` `none`
  The current transformation matrix to apply to an anchor's position before returning it. If `offset-anchor` and `default` is set, it will be first translated by the distance between them.
- **`name`** `str` `none`
  The name of the element, this is only used in the error message in the event an anchor is invalid.
- **`offset-anchor`** `str` `none`
  The name of an anchor to offset the transform by.
- **`border-anchors`** `bool`
  If true, add border anchors.
- **`path-anchors`** `bool`
  If true, add path anchors.
- **`radii`** `none` `array`
  Radius tuple used for border anchor calculation.
- **`path`** `none` `drawable`
  Path used for path and border anchor calculation.

Mark

check-mark

```
check-mark(
  style style
) → bool
```

Checks if a mark should be drawn according to the current style.

Parameters

- **`style`** `style`
  The current style.

process-style

```
process-style(
  ctx context,
  style style,
  root str,
  path-length float
)
```

Processes the mark styling. TODO: remember what is actually going on here.

Parameters

- **ctx** `context`
  The context object.
- **style** `style`
  The current style.
- **root** `str`
  Where the mark is being placed, normally either `"start"` or `"end"`. Allows different styling for marks in different directions.
- **path-length** `float`
  The length of the path. This is used for relative offsets.

place-mark-on-path

```
place-mark-on-path(
  ctx context,
  styles style,
  segments drawable,
  is-end: bool
) ⟶ dictionary
```

Places a mark on the given path. Returns a {{dictionary}} with the following keys:

—

Parameters

- **drawables** `drawable`
  The mark drawables.
- **distance** `float`
  The length to shorten the path by.
- **pos** `float`
  The position of the mark, can be used to snap the end of the path to after shortening.
- **ctx** `context`
  The canvas context object.
- **styles** `style`
  A processed mark styling.
- **segments** `drawable`
  The path to place the mark on.
- **is-end** `bool`
  Start from the end of the path

Result

`dictionary`

Dictionary with the following keys: pt, distance and drawable.

place-marks-along-path

```
place-marks-along-path(
  ctx context,
  style style,
  transform matrix,
  path drawable,
  add-path: bool
) ⟶ array
```

Places marks along a path. Returns them as an {{array}} of {{drawable}}.

Parameters

- **`ctx`** `context`
  The context object.
- **`style`** `style`
  The current mark styling.
- **`transform`** `matrix`
  The current transformation matrix.
- **`path`** `drawable`
  The path to place the marks on.
- **`add-path`** `bool`
  When `true` the shortened path will returned as the first {{drawable}} in the {{array}}

Bezier

quadratic-point

```
quadratic-point(
  a vector,
  b vector,
  c vector,
  t float
) ⟶ vector
```

Get the point on quadratic bezier at position `t`.

Parameters

- **`a`** `vector`
  Start point
- **`b`** `vector`
  End point
- **`c`** `vector`
  Control point
- **`t`** `float`
  Position on curve $[0, 1]$

quadratic-derivative

```
quadratic-derivative(
  a vector,
  b vector,
  c vector,
  t float
) ⟶ vector
```

Get the derivative (dx/dt) of a quadratic bezier at position `t`.

Parameters

- **`a`** `vector`
  Start point
- **`b`** `vector`
  End point
- **`c`** `vector`
  Control point
- **`t`** `float`
  Position on curve [0, 1]

cubic-point

```
cubic-point(
  a vector,
  b vector,
  c1 vector,
  c2 vector,
  t float
) ⟶ vector
```

Get the point on a cubic bezier curve at position `t`.

Parameters

- **`a`** `vector`
  Start point
- **`b`** `vector`
  End point
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2
- **`t`** `float`
  Position on curve [0, 1]

cubic-derivative

```
cubic-derivative(
  a vector,
  b vector,
  c1 vector,
  c2 vector,
```

```
  t float
) → vector
```

Get the derivative (dx/dt) of a cubic bezier at position `t`.

Parameters

- **`a`** `vector`
  Start point
- **`b`** `vector`
  End point
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2
- **`t`** `float`
  Position on curve [0, 1]

to-abc

```
to-abc(
  s vector,
  e vector,
  B vector,
  t float,
  deg: int
) → array
```

Get a bezier curve's ABC coordinates. Returns them as a respective array</Type> of vector</Type>s.

```
       /A\  <-- Control point of quadratic bezier
      / | \
     /  |  \
    /_.-B-._\  <-- Point on curve
  ,'    |    ',
 /      |      \
s------C------e  <-- Point on line between s and e
```

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`B`** `vector`
  Point on curve
- **`t`** `float`
  Position on curve [0, 1]
- **`deg`** `int`
  Bezier degree (2 or 3)

quadratic-through-3points

```
quadratic-through-3points(
  s vector,
  B vector,
  e vector
) ⟶ bezier
```

Compute the control points for a quadratic bezier through 3 points.

Parameters

- **s** `vector`
  Curve start
- **e** `vector`
  Curve end
- **B** `vector`
  A point which the curve passes through

quadratic-to-cubic

```
quadratic-to-cubic(
  s vector,
  e vector,
  c vector
) ⟶ bezier
```

Convert a quadratic bezier to a cubic bezier.

Parameters

- **s** `vector`
  Curve start
- **e** `vector`
  Curve end
- **c** `vector`
  Control point

cubic-through-3points

```
cubic-through-3points(
  s vector,
  B vector,
  e vector
) ⟶ bezier
```

Compute the control points for a cubic bezier through 3 points.

Parameters

- **s** `vector`
  Curve start
- **e** `vector`
  Curve end
- **B** `vector`
  A point which the curve passes through

split

```
split(
  s vector,
  e vector,
  c1 vector,
  c2 vector,
  t float
) ⟶ array
```

Split a cubic bezier into two cubic beziers at the point `t`. Returns an array</Type> of two bezier</Type>. The first holds the original curve start `s`, and the second holds the original curve end `e`.

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2
- **`t`** `float`
  The point on the bezier to split, $[0, 1]$

cubic-arclen

```
cubic-arclen(
  s vector,
  e vector,
  c1 vector,
  c2 vector,
  samples:
) ⟶ float
```

Get the approximate cubic curve length

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2

cubic-shorten-linear

```
cubic-shorten-linear(
  s vector,
  e vector,
```

```
  c1 vector,
  c2 vector,
  d float
) ⟶ bezier
```

Shorten the curve by offsetting s and c1 or e and c2 by distance d. If d is positive the curve gets shortened by moving s and c1 closer to e, if d is negative, e and c2 get moved closer to s.

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2
- **`d`** `float`
  Distance to shorten by

cubic-t-for-distance

```
cubic-t-for-distance(
  s vector,
  e vector,
  c1 vector,
  c2 vector,
  d float,
  samples:
) ⟶ float
```

Approximate bezier interval `t` for a given distance `d`. If `d` is positive, the functions starts from the curve's start `s`, if `d` is negative, it starts form the curve's end `e`.

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2
- **`d`** `float`
  The distance along the bezier to find `t`.

cubic-shorten

```
cubic-shorten(
  s vector,
  e vector,
```

```
  c1 vector,
  c2 vector,
  d float,
  samples: int
) ⟶ bezier
```

Shorten curve by distance `d`. This keeps the curvature of the curve by finding new values along the original curve. If `d` is positive the curve gets shortened by moving `s` closer to `e`, if `d` is negative, `e` is moved closer to `s`. The points `s` and `e` are moved along the curve, keeping the curve's curvature the same (the control points get recalculated).

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2
- **`d`** `float`
  Distance to shorten by
- **`samples`** `int`
  Maximum of samples/steps to use

cubic-extrema

```
cubic-extrema(
  s vector,
  e vector,
  c1 vector,
  c2 vector
) ⟶ array
```

Find cubic curve extrema by calculating the roots of the curve's first derivative. Returns an array</Type> of vector</Type> ordered by distance along the curve from the start to its end.

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2

cubic-aabb

```
cubic-aabb(
  s vector,
  e vector,
  c1 vector,
  c2 vector
) ⟶ array
```

Returns axis aligned bounding box coordinates `(bottom-left, top-right)` for a cubic bezier curve.

Parameters

- **`s`** `vector`
  Curve start
- **`e`** `vector`
  Curve end
- **`c1`** `vector`
  Control point 1
- **`c2`** `vector`
  Control point 2

catmull-to-cubic

```
catmull-to-cubic(
  points array,
  k float,
  close: bool
) ⟶ array
```

Returns an array of cubic bezier</Type> for a catmull curve through an array of points.

Parameters

- **`points`** `array`
  Array of 2d points
- **`k`** `float`
  Strength between 0 and 1
- **`close`** `bool`

line-cubic-intersections

```
line-cubic-intersections(
  la vector,
  lb vector,
  s vector,
  e vector,
  c1 vector,
  c2 vector,
  ray: bool
) ⟶ array
```

Calculate the intersection points between a 2D cubic-bezier and a straight line. Returns an array of vector</Type>

Parameters

- **`s`** `vector`
  Bezier start point
- **`e`** `vector`
  Bezier end point
- **`c1`** `vector`
  Bezier control point 1
- **`c2`** `vector`
  Bezier control point 2
- **`la`** `vector`
  Line start point
- **`lb`** `vector`
  Line end point
- **`ray`** `bool`
  If set to true, ignore line length

AABB

aabb

```
aabb(
  pts array,
  init: aabb
) ⟶ aabb
```

Compute an axis aligned bounding box (aabb) for a list of vectors</Type>.

Parameters

- **`pts`** `array`
  List of vector</Type>s.
- **`init`** `aabb`
  Initial aabb

mid

```
mid(
  bounds aabb
) ⟶ vector
```

Get the mid-point of an AABB as vector.

Parameters

- **`bounds`** `aabb`
  The AABB to get the mid-point of.

corner-points

```
corner-points(
  bounds aabb,
  front:
) ⟶ array
```

Get four corner points of the AABB as vectors.

Parameters

- **`bounds`** `aabb`

  The AABB

- **`fron`** `bool`

  Return the corner points for the high z component

Result

`array`

of vectors

size

```
size(
  bounds aabb
) ⟶ vector
```

Get the size of an aabb as vector. This is a vector from the aabb's low to high.

Parameters

- **`bounds`** `aabb`

  The aabb to get the size of.

padded

```
padded(
  bounds aabb,
  padding none dictionary
) ⟶ aabb
```

Pad AABB with padding from dictionary with keys top, left, right and bottom.

Parameters

- **`bounds`** `aabb`

  The AABB to pad.

- **`padding`** `none` `dictionary`

  Padding values

Hobby

hobby-to-cubic-open

```
hobby-to-cubic-open(
  points array,
  ta: auto array,
  tb: auto array,
  rho: auto function,
  omega: auto array
) ⟶ array
```

Calculates a bezier spline for an open Hobby curve through a list of points. Returns an {{array}} of {{bezier}}s

Parameters

- **`points`** `array`
  List of points
- **`ta`** `auto` `array`
  Outgoing tension per point
- **`tb`** `auto` `array`
  Incoming tension per point
- **`rho`** `auto` `function`
  The rho function of the form `(float, float) => float`
- **`omega`** `auto` `array`
  Tuple of the curl at the start end end of the curve `(start, end)` as floats

hobby-to-cubic-closed

```
hobby-to-cubic-closed(
  points array,
  ta: auto array,
  tb: auto array,
  rho: auto array
) ⟶ array
```

Calculates a bezier spline for a closed Hobby curve through a list of points. Returns an {{array}} of {{bezier}}s.

Parameters

- **`points`** `array`
  List of points
- **`ta`** `auto` `array`
  Outgoing tension per point
- **`tb`** `auto` `array`
  Incoming tension per point
- **`rho`** `auto` `array`
  The rho function of the form `(float, b) => float`

hobby-to-cubic

```
hobby-to-cubic(
  points array,
  ta: auto array,
  tb: auto array,
  rho: auto array,
  omega: auto array,
  close: bool
) ⟶ array
```

Calculates a bezier spline for a Hobby curve through a list of points. Returns an {{array}} of {{bezier}}s.

Parameters

- **`points`** `array`
  List of points

- **ta** `auto` `array`
  Outgoing tension per point
- **tb** `auto` `array`
  Incoming tension per point
- **rho** `auto` `array`
  The rho function of the form `(float, float) => float`
- **omega** `auto` `array`
  Tuple of the curl at the start end end of the curve `(start, end)` as floats
- **close** `bool`
  Close the curve

Intersection

line-line

```
line-line(
  p1 vector,
  p2 vector,
  p3 vector,
  p4 vector,
  ray: bool,
  eps: float
) → vector or none
```

Checks for a line-line intersection between the given points and returns its position, otherwise {{none}}.

Parameters

- **p1** `vector`
  Point 1
- **p2** `vector`
  Point 2
- **p3** `vector`
  Point 3
- **p4** `vector`
  Point 4
- **ray** `bool`
  If true, handle both lines as infinite rays
- **eps** `float`
  Epsilon

Result

- `vector`
  The intersection point between both lines
- `none`
  None, if both lines are parallel

line-cubic

```
line-cubic(
  la vector,
  lb vector,
  s vector,
  e vector,
  c1 vector,
  c2 vector
) ⟶ array
```

Finds the intersections of a line and cubic bezier.

Parameters

- **`s`** `vector`
  Bezier start point
- **`e`** `vector`
  Bezier end point
- **`c1`** `vector`
  Bezier control point 1
- **`c2`** `vector`
  Bezier control point 2
- **`la`** `vector`
  Line start point
- **`lb`** `vector`
  Line end point
- **`ray`** `bool`
  When `true`, intersections will be found for the whole line instead of inbetween the given points.

line-path

```
line-path(
  la vector,
  lb vector,
  path drawable
) ⟶ array
```

Finds the intersections of a line and path in 2D. The path should be given as a {{drawable}} of type `path`.

Parameters

- **`la`** `vector`
  Line start
- **`lb`** `vector`
  Line end
- **`path`** `drawable`
  The path.

path-path

```
path-path(
  a path,
```

```
  b path,
  samples: int
) ⟶ array
```

Finds the intersections between two path {{drawable}}s in 2D.

Parameters

- **a** `path`

  Path a
- **b** `path`

  Path b
- **samples** `int`

  Number of samples to use for bezier curves

Path Util

make-subpath

```
make-subpath(
  origin vector,
  segments array,
  closed: bool
) ⟶ subpath
```

Create a new subpath. A path is an array of subpaths.

Parameters

- **origin** `vector`

  Origin
- **segments** `array`

  Segments
- **closed** `bool`

  Closed

first-subpath-closed

```
first-subpath-closed(
  path
) ⟶ boolean
```

Get if the first subpath is closed

first-subpath-start

```
first-subpath-start(
  path
) ⟶ vector
```

Get the start position of the first path

subpath-start

```
subpath-start(
  subpath
) ⟶ vector
```

Get the start point of a subpath

subpath-end

```
subpath-end(
  subpath ,
  ignore-close-flag:
) ⟶ vector
```

Get the end point of a subpath

last-subpath-end

```
last-subpath-end(
  path
) ⟶ vector
```

Get the end position of the last path

bounds

```
bounds(
  path array
) ⟶ array
```

Calculates the bounding points for a list of path segments

Parameters

- **path** `array`

  Path

segment-lengths

```
segment-lengths(
  path path ,
  samples: auto int
) ⟶ array
```

Returns an array of arrays with the lengths of all path segments. One sub-array for each subpath and its segments.

Parameters

- **path** `path`

  Input path
- **samples** `auto` `int`

  Number of samples to use for curves

Result

`array`

Array of arrays of floats containing the segment lengths

length

```
length(
  segments path ,
  samples: auto int
) ⟶ float
```

Returns the sum of all segment lengths of a path.

Parameters

- **`segments`** `path`
  Path segments
- **`samples`** `auto` `int`
  Number of samples to take for curves

Result

`float`

Length

point-at

```
point-at(
  path path,
  distance ratio number,
  reverse: bool,
  samples: ,
  ignore-subpaths: bool
) ⟶ none or dictionary
```

Get information about a point at a given distance on a path.

Parameters

- **`path`** `path`
  The path
- **`distance`** `ratio` `number`
  Distance along the path
- **`reverse`** `bool`
  Travel from end to start
- **`ignore-subpaths`** `bool`
  If false consider the whole path, including sub-paths

Result

-
- `dictionary`
  Dictionary with the following keys: - point (vector) The point on the path - previous-point (vector) Point previous to point - direction (vector) Normalized direction vector - subpath-index (int) Index of the subpath - segment-index (int) Index of the segment None is returned, if the path is empty/of length zero.

shorten-to

```
shorten-to(
  path Path,
  distance number ratio array,
  reverse: boolean,
  mode: 'CURVED' 'LINEAR',
  samples: auto int,
  snap-to: none array
)
```

Shorten a path on one or both sides

Parameters

- **`path`** `Path`
  Path
- **`distance`** `number` `ratio` `array`
  Distance to shorten the path by
- **`reverse`** `boolean`
  If true, start from the end
- **`mode`** `'CURVED'` `'LINEAR'`
  Shortening mode for cubic segments
- **`samples`** `auto` `int`
  Samples to take for measuring cubic segments
- **`snap-to`** `none` `array`
  Optional array of points to try to move the shortened segment to

normalize

```
normalize(
  path path
) ⟶ path
```

Normalize a path:
- Add missing closing segments
- Remove zero-length line segments

Parameters

- **`path`** `path`
  Input path

Util

float-epsilon

```
float-epsilon(

)
```

Constant to be used as float rounding error

float-eq

```
float-eq(
  a float,
  b float,
  epsilon: float
) ⟶ bool
```

Compare two floating point numbers

Parameters

- **`a`** `float`
  First number
- **`b`** `float`
  Second number

- **epsilon** `float`

  Maximum distance between both numbers

apply-transform

```
apply-transform(
  transform matrix function,
  ..vecs vector
) → vector or array or dictionary
```

Multiplies vectors by a transformation matrix. If multiple vectors are given they are returned as an array, if only one vector is given only one will be returned, if a dictionary is given they will be returned in the dictionary with the same keys.

Parameters

- **transform** `matrix` `function`

  The 4×4 transformation matrix or a function that accepts and returns a vector.

- **..vecs** `vector`

  Vectors to get transformed. Only the positional part of the sink is used. A dictionary of vectors can also be passed and all will be transformed.

revert-transform

```
revert-transform(
  transform matrix,
  ..vecs
) → vector
```

Reverts the transform of the given vector

Parameters

- **transform** `matrix`

  Transformation matrix

- **vec** `vector`

  Vector to be transformed

line-pt

```
line-pt(
  a vector,
  b vector,
  t float
) → vector
```

Linearly interpolates between two points and returns its position

Parameters

- **a** `vector`

  Start point

- **b** `vector`

  End point

- **t** `float`

  Position on the line $[0, 1]$

line-normal

```
line-normal(
  a vector,
  b vector
) → vector
```

Get orthogonal vector to line

Parameters

- **a** `vector`

  Start point
- **b** `vector`

  End point

circle-arclen

```
circle-arclen(
  radius float,
  angle: angle
) → float
```

Calculates the arc-length of a circle or arc

Parameters

- **radius** `float`

  Circle or arc radius
- **angle** `angle`

  The angle of the arc.

ellipse-point

```
ellipse-point(
  center vector,
  radius float array,
  angle
) → vector
```

Get point on an ellipse for an angle

Parameters

- **center** `vector`

  Center
- **radius** `float` `array`

  Radius or tuple of x/y radii
- **angled** `angle`

  Angle to get the point at

calculate-circle-center-3pt

```
calculate-circle-center-3pt(
  a vector,
  b vector,
  c vector
) → vector
```

Calculates the center of a circle from 3 points. The z coordinate is taken from point a.

Parameters

- **a** `vector`
  Point 1
- **b** `vector`
  Point 2
- **c** `vector`
  Point 3

resolve-number

```
resolve-number(
  ctx context,
  num number
) ⟶ float
```

Converts a {{number}} to “canvas units”

Parameters

- **ctx** `context`
  The current context object.
- **num** `number`
  The number to resolve.

map-dict

```
map-dict(
  d,
  fn
) ⟶ dictionary
```

Call function `fn` for each key-value pair of `d` and return the transformed dictionary.

- d (dictionary) Input dictionary
- fn (function) Transformation function

resolve-radius

```
resolve-radius(
  radius number array
) ⟶ array
```

Ensures that a radius has an `x` and `y` component.

Parameters

- **radius** `number` `array`

min

```
min(
  ..a
) ⟶ float
```

Finds the minimum of a set of values while ignoring `none` values.

Parameters

- **a** `float` `none`

max

```
max(
  ..a float none
) ⟶ float
```

Finds the maximum of a set of values while ignoring `none` values.

Parameters

- **..a** `float` `none`

merge-dictionary

```
merge-dictionary(
  a dictionary,
  b dictionary,
  overwrite: bool
) ⟶ dictionary
```

Merges dictionary `b` onto dictionary `a`. If a key does not exist in `a` but does in `b`, it is inserted into `a` with `b`'s value. If a key does exist in `a` and `b`, the value in `b` is only inserted into `a` if the `overwrite` argument is `true`. If a key does exist both in `a` and `b` and both values are of type {{dictionary}} they will be recursively merged with this same function.

Parameters

- **a** `dictionary`
  Dictionary a
- **b** `dictionary`
  Dictionary b
- **overwrite** `bool`
  Whether to override an entry in `a` that also exists in `b` with the value in `b`.

measure

```
measure(
  ctx context,
  cnt content
) ⟶ vector
```

Measures the size of some {{content}} in canvas coordinates.

Parameters

- **ctx** `context`
  The current context object.
- **cnt** `content`
  The content to measure.

as-padding-dict

```
as-padding-dict(
  padding none number array dictionary
) ⟶ dictionary
```

Get a padding/margin dictionary with keys `(top, left, bottom, right)` from a padding value.

Type of `padding`:
`none` All sides padded by 0
`number` All sides are padded by the same value
`array` CSS like padding: `(y, x)`, `(top, x, bottom)` or `(top, right, bottom, left)`
`dictionary` Converts a Typst padding dictionary (top, left, bottom, right, x, y, rest) to a dictionary containing top, left, bottom and right.

Parameters

- **`padding`** `none` `number` `array` `dictionary`
  Padding specification

as-corner-radius-dict

```
as-corner-radius-dict(
  ctx context,
  radii none number dictionary,
  size none array
) ⟶ dictionary
```

Creates a corner-radius dictionary with keys `north-east`, `north-west`, `south-east` and `south-west` with values of a two element {{array}} of the radius in the `x` and `y` direction. Returns none if all radii are zero or none.

Parameters

- **`ctx`** `context`
  The current canvas context object
- **`radii`** `none` `number` `dictionary`
  The radius specification. A {{number}} will cause all corners to have the same radius. An {{array}} with two items will cause all corners to have the same rx and ry radius. A {{dictionary}} can be given where the key specifies the corner and the value specifies the radius. The value can be either {{number}} for a circle radius or {{array}} for an x and y radius. The keys `north`, `south`, `east` and `west` targets both corners in that cardinal direction e.g. `south` sets the south west and south east corners. The keys `north-east`, `north-west`, `south-east` and `south-west` targets the corresponding corner. The key `rest` targets all other corners that have not been target by other keys.
- **`size`** `none` `array`
  Tuple of `number` used to clamp the corner radii

sort-points-by-distance

```
sort-points-by-distance(
  base vector,
  pts array
) ⟶ array
```

Sorts an array of vectors by distance to a common position.

Parameters

- **`base`** `vector`
  The position to measure the distance of the other vectors from.

- **pts** `array`

  The array of vectors to sort.

resolve-stroke

```
resolve-stroke(
  stroke none stroke
) ⟶ dictionary
```

Resolves a stroke into a usable dictionary with all fields that are missing or auto set to their Typst defaults.

Parameters

- **stroke** `none` `stroke`

  The stroke to resolve.

assert-body

```
assert-body(
  body
)
```

Asserts whether a “body” has the correct type.