CeTZ

ein Typst Zeichenpaket

Johannes Wolf
fenjalien

Version 0.2.0

1 Introduction

This package provides a way to draw onto a canvas using a similar API 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!

The name CeTZ is a recursive acronym for "CeTZ, ein Typst Zeichenpaket" (german for "CeTZ, a Typst drawing package").

2 Usage

This is the minimal starting point:

```
#import "@preview/cetz:0.2.0"
#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 function's such as `line`.

2.1 CeTZ Unique Argument 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` Any coordinate system. See coordinate-systems.

`number` Any of `float`, `integer` or `length`.

`style` Named arguments (or a dictionary if used for a single argument) of style key-values.

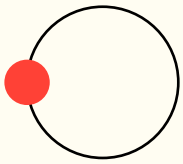
`context` A CeTZ context object that holds internal state.

`vector` A three element array of `float`s

2.2 Anchors

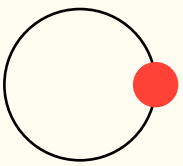
You can refer to a position relative to an element by using its anchors. Anchors come in several different variations but can all be used in two different ways.

The first is by using the `anchor` argument on an element. When given, the element will be translated such that the given anchor will be where the given position is. This is supported by all elements that have the `anchor` argument.



```
// Draw a circle and place its "west" anchor at the origin.
circle((0,0), anchor: "west")

// Draw a smaller red circle at the origin.
fill(red)
stroke(none)
circle((0,0), radius: 0.3)
```

The second is by using anchor coordinates. You must first give the element a name by passing a string to its `name` argument, you can then use its anchors to place other elements, see Section 4.6 for more usage. Note this is only available for elements that have a `name` argument.



```
// Name the circle
circle((0,0), name: "circle")

// Draw a smaller red circle at "circle"'s east anchor
fill(red)
stroke(none)
circle("circle.east", radius: 0.3)
```

Note that all anchors are transformed along with the element.

2.2.1 Named

Named anchors are normally unique to the type of element, such as a bezier curve's control points. Other anchor variants specify their own named anchors that are available to all elements that support the anchor variant.

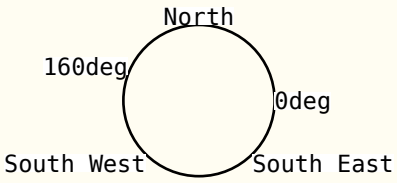
All elements also have a "default" named anchor, it always refers to another anchor on the element.

2.2.2 Border

A border anchor refers to a point on the element's border where a ray is cast from the element's center at a given angle and hits the border.

They are given as angles where `0deg` is towards the right and `90deg` is up.

Border anchors also specify named compass directions such as "north", "north-east", etc. Border anchors also spefcify a "center" named anchor which is where the ray cast originates from.

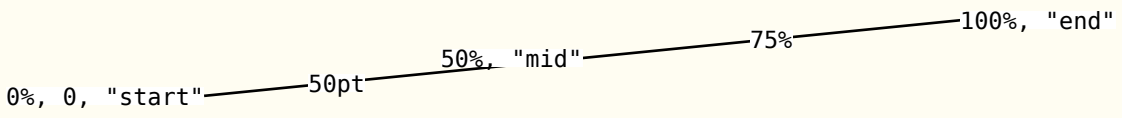



```
circle((0, 0), name: "circle", radius: 1)

content((name: "circle", anchor: 0deg), box(fill: white)[0deg], anchor: "west")
content((name: "circle", anchor: 160deg), box(fill: white)[160deg], anchor: "south-east")
content("circle.north", box(fill: white)[North], anchor: "south")
content("circle.south-east", box(fill: white)[South East], anchor: "north-west")
content("circle.south-west", box(fill: white)[South West], anchor: "north-east")
```

2.2.3 Path

A path anchor refers to a point along the path of an element. They can be given as either a `number` for an absolute distance along the path, or a `ratio` for a relative distance along the path.

Path anchors also specify three anchors "start", "mid" and "end".




```
line((0,0), (10, 1), name: "line")

content("line.start", box(fill: white)[0%, 0, "start"], anchor: "east")
content("line.mid", box(fill: white)[50%, "mid"], anchor: "east")
content("line.end", box(fill: white)[100%, "end"], anchor: "west")

content((name: "line", anchor: 75%), box(fill: white)[75%])
content((name: "line", anchor: 50pt), box(fill: white)[50pt])
```

3 Draw Function Reference

3.1 Canvas

3.1.1 canvas

Sets up a canvas for drawing on.

Parameters

canvas(
  length: `length` `ratio`,
  debug: `bool`,
  background: `none` `color`,
  body: `none` `array` `element`
) -> `content`

length: `length` or `ratio` Default: `1cm`

Used to specify what 1 coordinate unit is. If given a ratio, that ratio is relative to the containing elements width!

debug: `bool` Default: `false`

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

background: `none` or `color` Default: `none`

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

body: `none` or `array` or `element`

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

3.2 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`

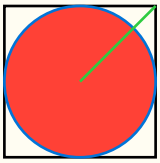
How to fill the drawn element.

stroke: `none` or `auto` or `length` or `color` or `dictionary` or `stroke` Default: `1pt + luma(0%)`

How to stroke the border or the path of the draw element. See Typst's line documentation for more details: https://typst.app/docs/reference/visualize/line/#parameters-stroke

```
// Draws a red circle with a blue border
circle((0, 0), fill: red, stroke: blue)
// Draws a green line
line((0, 0), (1, 1), stroke: green)
```

Instead of having to specify the same styling for each time you want to draw an element, you can use the `set-style()` function to change the style for all elements after it. You can still pass styling to a draw function to override what has been set with `set-style()`. You can also use the `fill()` and `stroke()` functions as a shorthand to set the fill and stroke respectively.



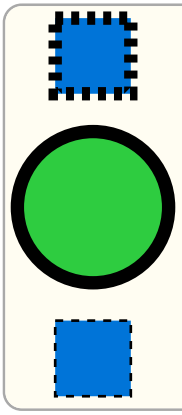
```
// Draws an empty square with a black border
rect((-1, -1), (1, 1))

// Sets the global style to have a fill of red and a stroke of blue
set-style(stroke: blue, fill: red)
circle((0,0))

// Draws a green line despite the global stroke is blue
line((), (1,1), stroke: green)
```

When using a dictionary for a style, it is important to note that they update each other instead of overriding the entire option like a non-dictionary value would do. For example, if the stroke is set to

`(paint: red, thickness: 5pt)` and you pass `(paint: blue)`, the stroke would become `(paint: blue, thickness: 5pt)`.

```
// Sets the stroke to red with a thickness of 5pt
set-style(stroke: (paint: red, thickness: 5pt))
// Draws a line with the global stroke
line((0,0), (1,0))
// Draws a blue line with a thickness of 5pt because dictionaries update the style
line((0,0), (1,1), stroke: (paint: blue))
// Draws a yellow line with a thickness of 1pt because other values override the style
line((0,0), (0,1), stroke: yellow)
```

You can also specify styling for each type of element. Note that dictionary values will still update with its global value, the full hierarchy is `function > element type > global`. When the value of a style is `auto`, it will become exactly its parent style.



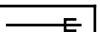
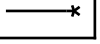
```
set-style(
  // Global fill and stroke
  fill: green,
  stroke: (thickness: 5pt),
  // Stroke and fill for only rectangles
  rect: (stroke: (dash: "dashed"), fill: blue),
)
rect((0,0), (1,1))
circle((0.5, -1.5))
rect((0,-3), (1, -4), stroke: (thickness: 1pt))
```

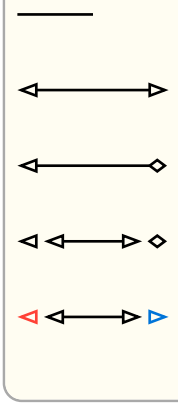
```
// Its a nice drawing okay
set-style(
  rect: (
    fill: red,
    stroke: none
  ),
  line: (
    fill: blue,
    stroke: (dash: "dashed")
  ),
)
rect((0,0), (1,1))
line((0, -1.5), (0.5, -0.5), (1, -1.5), close: true)
circle((0.5, -2.5), radius: 0.5, fill: green)
```

3.2.1 Marks

Marks are arrow tips that can be added to the end of path based elements that support the `mark` style key, or can be directly drawn by using the `mark` draw function. Marks are specified by giving there names as strings and have several options to customise them. You can give an array of names to have multiple marks in a row, dictionaries can also be used in the array for per mark styling.

Name	Shorthand	Shape
triangle	>	
triangle (reversed)	<	
diamond	<>	
rect	[]	
bracket	]	

bracket (reversed)	[	
bar	|	
circle	o	
plus	+	
x	x	
star	*	



```typ
let c = ((rel: (0, -1)), (rel: (2, 0), update: false)) // Coordinates to draw the line, it is not necessary to understand this for this example.
// No marks
line((), (rel: (1, 0), update: false))
// Draws a triangle mark at both ends of the line.
set-style(mark: (symbol: ">"))
line(..c)
// Overrides the end mark to be a diamond but the start is still a triangle.
set-style(mark: (end: "<>"))
line(..c)
// Draws two triangle marks at both ends but the first mark of end is still a diamond.
set-style(mark: (symbol: (">", ">")))
line(..c)
// Sets the stroke of first mark in the sequence to red but the end mark overrides it to be blue.
set-style(mark: (symbol: ((symbol: ">", stroke: red), ">"), end: (stroke: blue)))
line(..c)
```

symbol: `none` or `string` or `array` or `dictionary` Default: `none`

This option sets the mark to draw when using the `mark` draw function, or applies styling to both mark ends of path based elements. The mark's name or shorthand can be given, multiple marks can be drawn by passing an array of names or shorthands. When `none` no marks will be drawn. A style `dictionary` can be given instead of a `string` to override styling for that particular mark, just make sure to still give the mark name using the `symbol` key otherwise nothing will be drawn!.

start: `none` or `string` or `array` or `dictionary` Default: `none`

This option sets the mark to draw at the start of a path based element. It will override all options of the `symbol` key and will not effect marks drawn using the `mark` draw function.

end: `none` or `string` or `array` or `dictionary` Default: `none`

This option sets the mark to draw at the end of a path based element. It will override all options of the `symbol` key and will not effect marks drawn using the `mark` draw function.

length: `number` Default: `5.67pt`

The size of the mark in the direction it is pointing.

width: `number` Default: `4.25pt`

The size of the mark along the normal of its direction.

inset: `number` Default: `1.42pt`

It specifies a distance by which something inside the arrow tip is set inwards; for the Stealth arrow tip it is the distance by which the back angle is moved inwards.

scale: `float` Default: `1`

A factor that is applied to the mark's length, width and inset.

sep: `number` Default: `2.83pt`

The distance between multiple marks along their path.

flex: `boolean` Default: `true`

Only applicable when marks are used on curves such as bezier and hobby. If true, the mark will point along the secant of the curve. If false, the tangent at the marks tip is used.

position-samples: `integer` Default: `30`

Only applicable when marks are used on curves such as bezier and hobby. The maximum number of samples to use for calculating curve positions. A higher number gives better results but may slow down compilation.

pos: `number` or `ratio` Default: `none`

Overrides the mark's position along a path. A number will move it an absolute distance, while a ratio will be a distance relative to the length of the path. Note that this may be removed in the future in preference of a different method.

offset: `number` or `ratio` Default: `none`

Like `pos` but it moves the position of the mark instead of overriding it.

slant: `ratio` Default: `0%`

How much to slant the mark relative to the axis of the arrow. 0% means no slant 100% slants at 45 degrees

harpoon: `boolean` Default: `false`

When true only the top half of the mark is drawn.

flip: `boolean` Default: `false`

When true the mark is flipped along its axis.

reverse: `boolean` Default: `false`

Reverses the direction of the mark.

xy-up: `vector` Default: `(0, 0, 1)`

The direction which is "up" for use when drawing 2D marks.

z-up: `vector` Default: `(0, 1, 0)`

The direction which is "up" for use when drawing 3D marks.

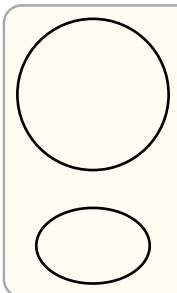
shorten-to: `integer` or `auto` or `none` Default: `none`

Which mark to shorten the path to when multiple marks are given. `auto` will shorten to the last mark, `none` will shorten to the first mark (effectively disabling path shortening). An integer can be given to select the mark's index.

3.3 Shapes

3.3.1 circle

Draws a circle or ellipse.



```
circle((0,0))
// Draws an ellipse
circle((0,-2), radius: (0.75, 0.5))
```

Parameters

```
circle(
  position: coordinate,
  name: none string,
  anchor: none string,
  ..style: style
)
```

position: `coordinate`

The position to place the circle on.

Styling

Root: `circle`

Keys

radius: `number` or `array` Default: `1`

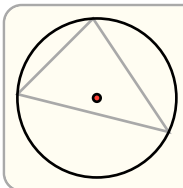
A number that defines the size of the circle's radius. Can also be set to a tuple of two numbers to define the radii of an ellipse, the first number is the `x` radius and the second is the `y` radius.

Anchors

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

3.3.2 circle-through

Draws a circle through three coordinates.



```
let (a, b, c) = ((0,0), (2,-.5), (1,1))
line(a, b, c, close: true, stroke: gray)
circle-through(a, b, c, name: "c")
circle("c.center", radius: .05, fill: red)
```

Parameters

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

a: `coordinate`

Coordinate a.

b: `coordinate`

Coordinate b.

c: `coordinate`

Coordinate 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.

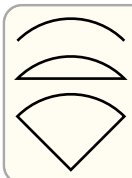
Anchors

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

- **a** Coordinate a
- **b** Coordinate b
- **c** Coordinate c

3.3.3 arc

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`).

Parameters

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

position: `coordinate`

Position to place the arc at.

start: `auto` or `angle` Default: `auto`

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

stop: `auto` or `angle` Default: `auto`

The angle at which the arc should stop.

delta: `auto` or `angle` Default: `auto`

The change in angle away start or stop.

Styling

Root: `arc`

Keys

radius: `number` or `array` Default: `1`

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: `string` Default: `"OPEN"`

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` Default: `true`

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.

Anchors

Supports border and path anchors.

center The center of the arc, this is the default anchor.
arc-center The midpoint of the arc’s curve.
chord-center Center of chord of the arc drawn between the start and end point.
origin The origin of the arc’s circle.
arc-start The position at which the arc’s curve starts.
arc-end The position of the arc’s curve end.

3.3.4 arc-through

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.

```
arc-through((0,1), (1,1), (1,0))
```

Parameters

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

a: `coordinate`

Start position of the arc

b: `coordinate`

Position the arc passes through

c: `coordinate`

End position of the arc

Styling

Root: `arc`

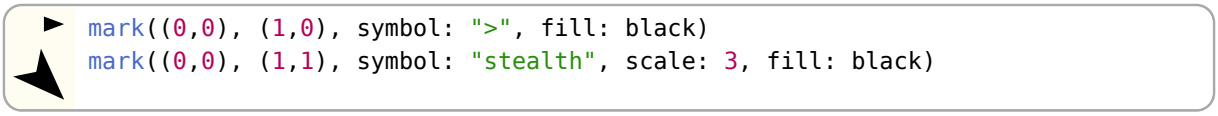
Uses the same styling as `arc()`

Anchors

For anchors see `arc()`.

3.3.5 mark

Draws a single mark pointing towards a target coordinate.



```
mark((0,0), (1,0), symbol: ">", fill: black)
mark((0,0), (1,1), symbol: "stealth", scale: 3, fill: black)
```

Parameters

mark(
  from: `coordinate`,
  to: `coordinate`,
  ..style: `style`
)

from: `coordinate`

The position to place the mark.

to: `coordinate`

The position the mark should point towards.

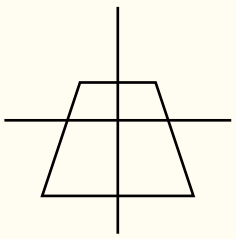
Styling

Root: `mark`

You can directly use the styling from Section 3.2.1.

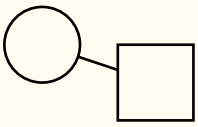
3.3.6 line

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



```
line((-1.5, 0), (1.5, 0))
line((0, -1.5), (0, 1.5))
line((-1, -1), (-0.5, 0.5), (0.5, 0.5), (1, -1), 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")
```

Parameters

```
line(
  ..pts-style: coordinates style,
  close: bool,
  name: none string
)
```

..pts-style: `coordinates` or `style`

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

close: `bool` Default: `false`

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

Styling

Root: `line`

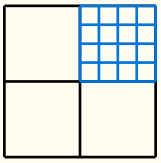
Supports mark styling.

Anchors

Supports path anchors.

3.3.7 grid

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)
```

Parameters

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

from: `coordinate`

The top left of the grid

to: `coordinate`

The bottom right of the grid

Styling

Root: `grid`

Keys

step: `number` or `tuple` or `dictionary` Default: `1`

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.

help-lines: `bool` Default: `1`

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

Anchors

Supports border anchors.

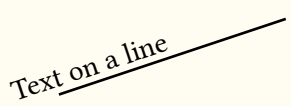
3.3.8 content

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

Hello World!

```
content((0,0), [Hello World!])
```

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`:




```
line((0, 0), (3, 1), name: "line")
content(
  ("line.start", 0.5, "line.end"),
  angle: "line.end",
  padding: .1,
  anchor: "south",
  [Text on a line]
)
```

This is a long text.

```
// Place content in a rect between two coordinates
content((0, 0), (2, 2), box(par(justify: false)[This is a long text.],
stroke: 1pt, width: 100%, height: 100%, inset: 1em))
```

Parameters

```
content(
  ..args-style: coordinate content style,
  angle: angle coordinate,
  anchor: none string,
  name: none string
)
```

..args-style: `coordinate` or `content` or `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` or `coordinate` Default: `0deg`

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.

Styling

Root: `content`

Keys

padding: `number` or `dictionary` Default: `0`

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: `string` or `none` Default: `none`

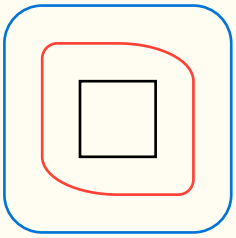
Sets the frame style. Can be `none`, "rect" or "circle" and inherits the `stroke` and `fill` style.

Anchors

Supports border anchors.

3.3.9 rect

Draws a rectangle between two coordinates.



```
rect((0,0), (1,1))
rect((-.5, -.5), (rel: (2, 2)),
  radius: (north-east: (100%, .5),
           south-west: (100%, .5), rest: .2),
  stroke: red)
rect((-1, -1), (rel: (3, 3)), radius: .5, stroke: blue)
```

Parameters

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

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))`.

Styling

Root `rect`

Keys

radius: `number` or `ratio` or `dictionary` Default: `0`

The rectangles 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, 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 rects width/height.

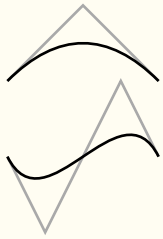
```
rect((0,0), (rel: (1,1)), radius: 0)
rect((2,0), (rel: (1,1)), radius: 25%)
rect((4,0), (rel: (1,1)), radius: (north: 50%))
rect((6,0), (rel: (1,1)), radius: (north-east: 50%))
rect((8,0), (rel: (1,1)), radius: (south-west: 0, rest: 50%))
rect((10,0), (rel: (1,1)), radius: (rest: (20%, 50%)))
```

Anchors

Supports border and path anchors.

3.3.10 bezier

Draws a quadratic or cubic bezier curve



```
let (a, b, c) = ((0, 0), (2, 0), (1, 1))
line(a, c,  b, stroke: gray)
bezier(a, b, c)

let (a, b, c, d) = ((0, -1), (2, -1), (.5, -2), (1.5, 0))
line(a, c, d, b, stroke: gray)
bezier(a, b, c, d)
```

Parameters

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

start: `coordinate`

Start position

end: `coordinate`

End position (last coordinate)

..ctrl-style: `coordinate` or `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.

Styling

Root `bezier`

Supports marks.

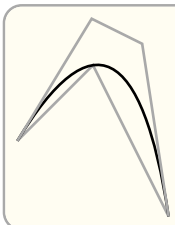
Anchors

Supports path anchors.

ctrl-n nth control point where n is an integer starting at 0

3.3.11 bezier-through

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))
line(a, b, c, stroke: gray)
bezier-through(a, b, c, name: "b")

// Show calculated control points
line(a, "b.ctrl-0", "b.ctrl-1", c, stroke: gray)
```

Parameters

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

start: `coordinate`

The position to start the curve.

pass-through: `coordinate`

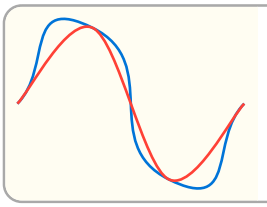
The position to pass the curve through.

end: `coordinate`

The position to end the curve.

3.3.12 catmull

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)
```

Parameters

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

..pts-style: `coordinate` or `style`

Positional arguments should be coordinates that the curve should pass through. Named arguments are for styling.

close: `bool` Default: `false`

Closes the curve with a straight line between the start and end of the curve.

Styling

Root `catmull`

Supports marks.

Keys

tension: `float` Default: `0.5`

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

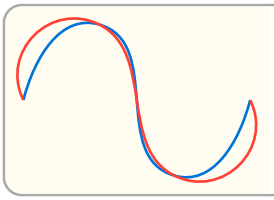
Anchors

Supports path anchors.

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

3.3.13 hobby

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)
```

Parameters

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

..pts-style: `coordinate` or `style`

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

ta: `auto` or `array` Default: `auto`

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.

tb: `auto` or `array` Default: `auto`

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.

close: `bool` Default: `false`

Closes the curve with a proper smooth curve between the start and end of the curve.

Styling

Root `hobby`

Supports marks.

Keys

omega: `tuple of float` Default: `(1, 1)`

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.

Anchors

Supports path anchors.

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

3.3.14 merge-path

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()` are ignored.

Parameters

```
merge-path(
  body: elements,
  close: bool,
  name: none string,
  ..style: style
)
```

body: `elements`

Elements with paths to be merged together.

close: `bool` Default: `false`

Close the path with a straight line from the start of the path to its end.

Anchors

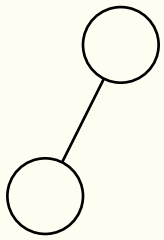
Supports path anchors.

3.4 Grouping

3.4.1 hide

Hides an element.

Hidden elements are not drawn to the canvas, are ignored when calculating bounding boxes and discarded by `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

hide(body: `element`)

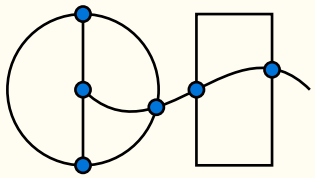
body: `element`

One or more elements to hide

3.4.2 intersections

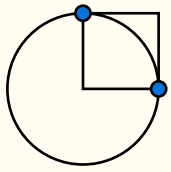
Calculates the intersections between multiple paths and creates one anchor per intersection point.

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()`.

Parameters

intersections(
  name: `string`,
  ..elements: `elements` `string`,
  samples: `int`
)

name: `string`

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

..elements: `elements` or `string`

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` Default: `10`

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

3.4.3 group

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))
```

Parameters

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

body: `elements` or `function`

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

anchor: `none` or `string` Default: `none`

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

Styling

Root `group`

Keys

padding: `none` or `number` or `array` or `dictionary` Default: `none`

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)`.

Anchors

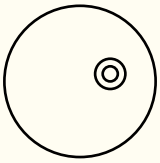
Supports compass, distance and angle anchors. However they are created based on the axis aligned bounding box of all the child elements of the group.

You can add custom anchors to the group by using the `anchor` element while in the scope of said group, see `anchor` for more details. You can also copy over anchors from named child element by using the `copy-anchors` element as they are not accessible from outside the group.

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

3.4.4 anchor

Creates a new anchor for the current group. This element can only be used inside a group otherwise it will panic. The new anchor will be accessible from inside the group by using just the anchor's name as a coordinate.



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

Parameters

```
anchor(
  name: string,
  position: coordinate
)
```

name: `string`

The name of the anchor

position: `coordinate`

The position of the anchor

3.4.5 copy-anchors

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

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

element: `string`

The name of the element to copy anchors from.

filter: `auto` or `array` Default: `auto`

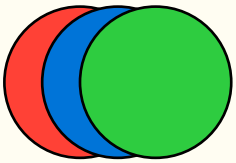
When set to `auto` all anchors will be copied to the group. An array of anchor names can instead be given so only the anchors that are in the element and the list will be copied over.

3.4.6 set-ctx

An advanced element that allows you to modify the current canvas context.

A context object holds the canvas' state, such as the element dictionary, the current transformation matrix, group and canvas unit length. The following fields are considered stable:

- `length` (length): Length of one canvas unit as typst length
- `transform` (cetz.matrix): Current 4x4 transformation matrix
- `debug` (bool): True if the canvas' debug flag is set



```
// 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)
```

Parameters

set-ctx(callback: `function`)

callback: `function`

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

3.4.7 get-ctx

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

```
(
  (1, 0, 0.5, 0),
  (0, −1, −0.5, 0),
  (0, 0, 1, 0),
  (0, 0, 0, 1),
)
```

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

Parameters

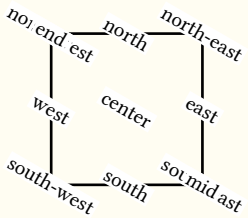
get-ctx(callback: `function`)

callback: `function`

A function that accepts the context dictionary and can return elements.

3.4.8 for-each-anchor

Iterates through all 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", (name) => {
   content((), box(inset: 1pt, fill: white, text(8pt, [#name])),
angle: -30deg)
})
```

Parameters

for-each-anchor(
  name: `string`,
  callback: `function`
)

name: `string`

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

callback: `function`

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

3.4.9 on-layer

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

```
on-layer(
  layer: float integer,
  body: elements
)
```

layer: `float` or `integer`

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

body: `elements`

Elements to draw on the layer specified.

3.5 Transformations

All transformation functions push a transformation matrix onto the current transform stack. To apply transformations scoped use a `group(...)` object.

Transformation matrices get multiplied in the following order:

$$M_{\text{world}} = M_{\text{world}} \cdot M_{\text{local}}$$

3.5.1 set-transform

Sets the transformation matrix.

Parameters

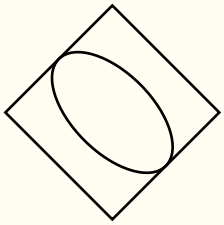
set-transform(mat: `none` `matrix`)

mat: `none` or `matrix`

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

3.5.2 rotate

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

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

..angles: `angle`

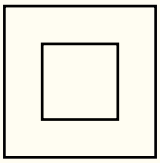
A single angle as a positional argument to rotate on the z-axis by. Named arguments of `x`, `y` or `z` can be given to rotate on their respective axis. You can give named arguments of `yaw`, `pitch` or `roll`, too.

origin: `none` or `coordinate` Default: `none`

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

3.5.3 translate

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

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

..args: `vector` or `float` or `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`.

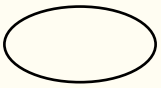
pre: `bool` Default: `false`

Specify matrix multiplication order

- false: `World = World * Translate`
- true: `World = Translate * World`

3.5.4 scale

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



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

Parameters

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

..args: `float` or `ratio`

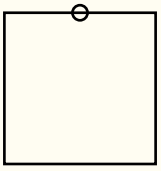
A single value to scale the transformation matrix by or per axis scaling factors. Accepts a single float or ratio value or any combination of the named arguments `x`, `y` and `z` to set per axis scaling factors. A ratio of 100% is the same as the value 1.

origin: `none` or `coordinate` Default: `none`

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

3.5.5 set-origin

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



```
// Outer rect
rect((0,0), (2,2), name: "r")
// Move origin to top edge
set-origin("r.north")
circle((0, 0), radius: .1)
```

Parameters

set-origin(origin: `coordinate`)

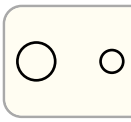
origin: `coordinate`

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

3.5.6 move-to

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

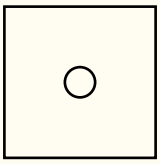
move-to(pt: `coordinate`)

pt: `coordinate`

The coordinate to move to.

3.5.7 set-viewport

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

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

from: `coordinate`

Bottom-Left corner coordinate

to: `coordinate`

Top right corner coordinate

bounds: `vector` Default: `(1, 1, 1)`

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

4 Coordinate Systems

A *coordinate* is a position on the canvas on which the picture is drawn. They take the form of dictionaries and the following sub-sections define the key value pairs for each system. Some systems have a more implicit form as an array of values and `CeTZ` attempts to infer the system based on the element types.

4.1 XYZ

Defines a point `x` units right, `y` units upward, and `z` units away.

x: `number` Default: `0`

The number of units in the `x` direction.

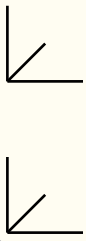
y: `number` Default: `0`

The number of units in the `y` direction.

z: `number` Default: `0`

The number of units in the `z` direction.

The implicit form can be given as an array of two or three `number`s, as in `(x,y)` and `(x,y,z)`.

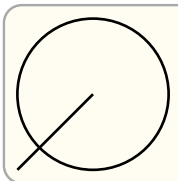


```
line((0,0), (x: 1))
line((0,0), (y: 1))
line((0,0), (z: 1))

// Implicit form
line((0, -2), (1, -2))
line((0, -2), (0, -1, 0))
line((0, -2), (0, -2, 1))
```

4.2 Previous

Use this to reference the position of the previous coordinate passed to a draw function. This will never reference the position of a coordinate used in to define another coordinate. It takes the form of an empty array `()`. The previous position initially will be `(0, 0, 0)`.



```
line((0,0), (1, 1))

// Draws a circle at (1,1)
circle(())
```

4.3 Relative

Places the given coordinate relative to the previous coordinate. Or in other words, for the given coordinate, the previous coordinate will be used as the origin. Another coordinate can be given to act as the previous coordinate instead.

rel: `coordinate`

The coordinate to be place relative to the previous coordinate.

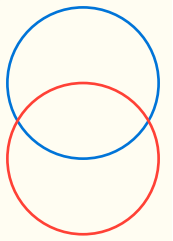
update: `boolean` Default: `true`

When false the previous position will not be updated.

to: `coordinate` Default: `()`

The coordinate to treat as the previous coordinate.

In the example below, the red circle is placed one unit below the blue circle. If the blue circle was to be moved to a different position, the red circle will move with the blue circle to stay one unit below.



```
circle((0, 0), stroke: blue)
circle((rel: (0, -1)), stroke: red)
```

4.4 Polar

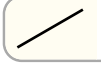
Defines a point that is `radius` distance away from the origin at the given `angle`.

angle: `angle`

The angle of the coordinate. An angle of `0deg` is to the right, a degree of `90deg` is upward. See https://typst.app/docs/reference/layout/angle/ for details.

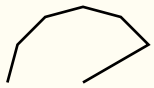
radius: `number` or `tuple<number>`

The distance from the origin. An array can be given, in the form `(x, y)` to define the `x` and `y` radii of an ellipse instead of a circle.



```
line((0,0), (angle: 30deg, radius: 1cm))
```

The implicit form is an array of the angle then the radius `(angle, radius)` or `(angle, (x, y))`.



```
line((0,0), (30deg, 1), (60deg, 1),
     (90deg, 1), (120deg, 1), (150deg, 1), (180deg, 1))
```

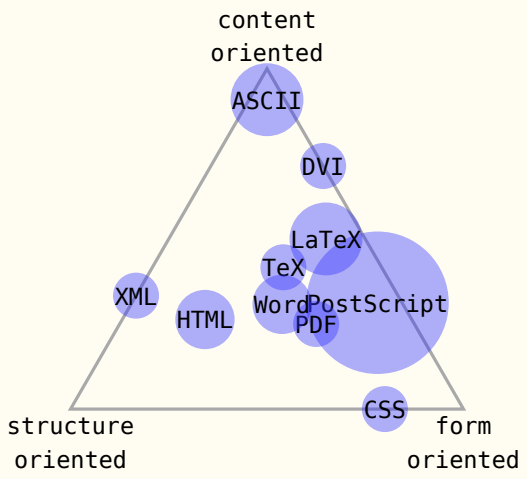
4.5 Barycentric

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors v_1, v_2 ..., v_n and numbers α_1, α_2, ..., α_n. Then the barycentric coordinate specified by these vectors and numbers is

$$\frac{\alpha_1 v_1 + \alpha_2 v_1 + \cdots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \cdots + \alpha_n}$$

bary: `dictionary`

A dictionary where the key is a named element and the value is a `float`. The `center` anchor of the named element is used as v and the value is used as a.




```
circle((90deg, 3), radius: 0, name: "content")
circle((210deg, 3), radius: 0, name: "structure")
circle((-30deg, 3), radius: 0, name: "form")

for (c, a) in (
  ("content", "south"),
  ("structure", "north"),
  ("form", "north")
) {
  content(c, align(center, c + [\ oriented]), padding: .
1, anchor: a)
}

stroke(gray + 1.2pt)
line("content", "structure", "form", close: true)

for (c, s, f, cont) in (
  (0.5, 0.1, 1, "PostScript"),
  (1, 0, 0.4, "DVI"),
  (0.5, 0.5, 1, "PDF"),
  (0, 0.25, 1, "CSS"),
  (0.5, 1, 0, "XML"),
  (0.5, 1, 0.4, "HTML"),
  (1, 0.2, 0.8, "LaTeX"),
  (1, 0.6, 0.8, "TeX"),
  (0.8, 0.8, 1, "Word"),
  (1, 0.05, 0.05, "ASCII")
) {
  content((bary: (content: c, structure: s, form: f)),
      cont, fill: rgb(50, 50, 255, 100), stroke: none,
frame: "circle")
}
```

4.6 Anchor

Defines a point relative to a named element using anchors, see Section 2.2.

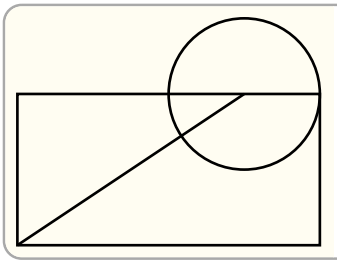
name: `string`

The name of the element that you wish to use to specify a coordinate.

anchor: `number` or `angle` or `string` Default: `none`

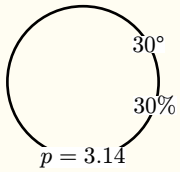
An anchor of the element. If one is not given a default anchor will be used. On most elements this is `center` but it can be different.

You can also use implicit syntax of a dot separated string in the form `"name.anchor"` for named anchors.



```
line((0,0), (3,2), name: "line")
circle("line.end", name: "circle")
rect("line.start", "circle.east")
```

Using the dictionary anchor syntax, you can not only use named anchors, but also query the element for distance or angle anchors on it's path:




```
circle((0,0), name: "circle")
// Anchor at 30 degree
content((name: "circle", anchor: 30deg), box(fill: white, $ 30 degree $))
// Anchor at 30% of the path length
content((name: "circle", anchor: 30%), box(fill: white, $ 30 % $))
// Anchor at 3.14 of the path
content((name: "circle", anchor: 3.14), box(fill: white, $ p = 3.14 $))
```

Note, that not all elements provide angle or distance based anchors!

4.7 Tangent

This system allows you to compute the point that lies tangent to a shape. In detail, consider an element and a point. Now draw a straight line from the point so that it "touches" the element (more formally, so that it is *tangent* to this element). The point where the line touches the shape is the point referred to by this coordinate system.

element: `string`

The name of the element on whose border the tangent should lie.

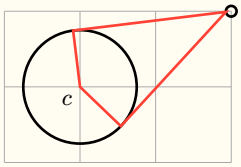
point: `coordinate`

The point through which the tangent should go.

solution: `integer`

Which solution should be used if there are more than one.

A special algorithm is needed in order to compute the tangent for a given shape. Currently it does this by assuming the distance between the center and top anchor (See Section 2.2) is the radius of a circle.




```
grid((0,0), (3,2), help-lines: true)

circle((3,2), name: "a", radius: 2pt)
circle((1,1), name: "c", radius: 0.75)
content("c", $ c $, anchor: "north-east", padding: .1)

stroke(red)
line("a", (element: "c", point: "a", solution: 1),
     "c", (element: "c", point: "a", solution: 2),
     close: true)
```

4.8 Perpendicular

Can be used to find the intersection of a vertical line going through a point p and a horizontal line going through some other point q.

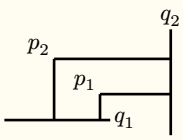
horizontal: `coordinate`

The coordinate through which the horizontal line passes.

vertical: `coordinate`

The coordinate through which the vertical line passes.

You can use the implicit syntax of `(horizontal, "-|", vertical)` or `(vertical, "|-", horizontal)`




```
set-style(content: (padding: .05))
content((30deg, 1), $ p_1 $, name: "p1")
content((75deg, 1), $ p_2 $, name: "p2")

line((-0.2, 0), (1.2, 0), name: "xline")
content("xline.end", $ q_1 $, anchor: "west")
line((2, -0.2), (2, 1.2), name: "yline")
content("yline.end", $ q_2 $, anchor: "south")

line("p1.south-east", (horizontal: (), vertical: "xline.end"))
line("p2.south-east", ((), "|-", "xline.end")) // Short form
line("p1.south-east", (vertical: (), horizontal: "yline.end"))
line("p2.south-east", ((), "-|", "yline.end")) // Short form
```

4.9 Interpolation

Use this to linearly interpolate between two coordinates `a` and `b` with a given distance `number`. If `number` is a `number` the position will be at the absolute distance away from `a` towards `b`, a `ratio` can be given instead to be the relative distance between `a` and `b`. An angle can also be given for the general meaning: "First consider the line from `a` to `b`. Then rotate this line by `angle` around point `a`. Then the two endpoints of this line will be `a` and some point `c`. Use this point `c` for the subsequent computation."

a: `coordinate`

The coordinate to interpolate from.

b: `coordinate`

The coordinate to interpolate to.

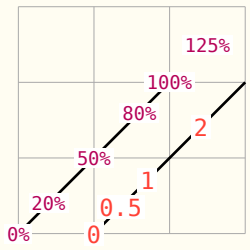
number: `ratio` or `number`

The distance between `a` and `b`. A ratio will be the relative distance between the two points, a number will be the absolute distance between the two points.

angle: `angle` Default: `0deg`

Angle between $\overrightarrow{AB}$ and $\overrightarrow{AP}$, where P is the resulting coordinate. This can be used to get the *normal* for a tangent between two points.

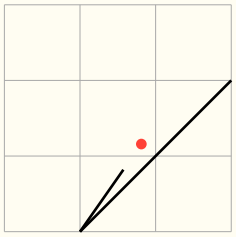
Can be used implicitly as an array in the form `(a, number, b)` or `(a, number, angle, b)`.



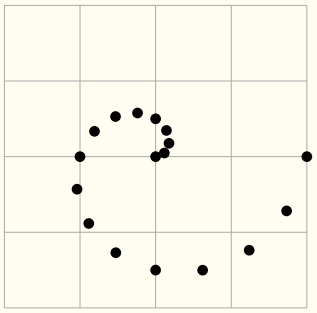

```
grid((0,0), (3,3), help-lines: true)

line((0,0), (2,2), name: "a")
for i in (0%, 20%, 50%, 80%, 100%, 125%) { /* Relative distance */
  content(("a.start", i, "a.end"),
          box(fill: white, inset: 1pt, [#i]))
}

line((1,0), (3,2), name: "b")
for i in (0, 0.5, 1, 2) { /* Absolute distance */
  content(("b.start", i, "b.end"),
          box(fill: white, inset: 1pt, text(red, [#i])))
}
```



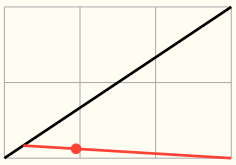
```
grid((0,0), (3,3), help-lines: true)
line((1,0), (3,2))
line((1,0), ((1, 0), 1, 10deg, (3,2)))
fill(red)
stroke(none)
circle(((1, 0), 50%, 10deg, (3, 2)), radius: 2pt)
```



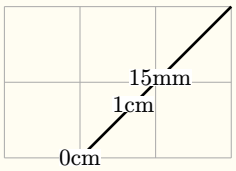
```
grid((0,0), (4,4), help-lines: true)

fill(black)
stroke(none)
let n = 16
for i in range(0, n+1) {
  circle(((2,2), i / 8, i * 22.5deg, (3,2)), radius: 2pt)
}
```

You can even chain them together!

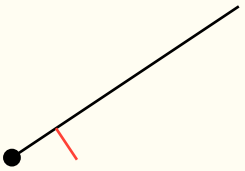


```
grid((0,0), (3, 2), help-lines: true)
line((0,0), (3,2))
stroke(red)
line(((0,0), 0.3, (3,2)), (3,0))
fill(red)
stroke(none)
circle(
  ( // a
    (((0, 0), .3, (3, 2))),
    0.7,
    (3,0)
  ),
  radius: 2pt
)
```




```
grid((0,0), (3, 2), help-lines: true)
line((1,0), (3,2))
for (l, c) in ((0cm, "0cm"), (1cm, "1cm"), (15mm, "15mm")) {
  content(((1,0), l, (3,2)), box(fill: white, $ #c $))
}
```

Interpolation coordinates can be used to get the *normal* on a tangent:

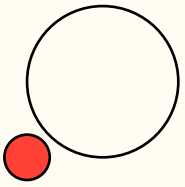


```
let (a, b) = ((0,0), (3,2))
line(a, b)
// Get normal for tangent from a to () with distance .5, at a
circle(a, radius: .1, fill: black)
line((a, .7, b), (a: (), b: a, number: .5, angle: 90deg), stroke: red)
```

4.10 Function

An array where the first element is a function and the rest are coordinates will cause the function to be called with the resolved coordinates. The resolved coordinates have the same format as the implicit form of the 3-D XYZ coordinate system, Section 4.1.

The example below shows how to use this system to create an offset from an anchor, however this could easily be replaced with a relative coordinate with the `to` argument set, Section 4.3.



```
circle((0, 0), name: "c")
fill(red)
circle((v => cetz.vector.add(v, (0, -1)), "c.west"), radius: 0.3)
```

5 Libraries

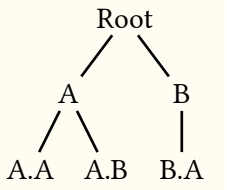
5.1 Tree

The tree library allows the drawing diagrams with simple tree layout algorithms

5.1.1 tree

Lays out and renders tree nodes.

For each node, the `tree` function creates an anchor of the format `"node-<depth>-<child-index>"` 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

```
tree(
  root: array,
  draw-node: auto function,
  draw-edge: none auto function,
  direction: string,
  parent-position: string,
  grow: float,
  spread: float,
  name: none string
)
```

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` or `function` Default: `auto`

The function to call to draw a node. The function will be passed two positional arguments, the node to draw and the node's parent, and is expected to return elements (`(node, parent-node) => elements`). The node's position is accessible through the "center" anchor or by using the previous position coordinate `()`. If `auto` is given, just the node's value will be drawn as content. The following predefined styles can be used:

draw-edge: `none` or `auto` or `function` Default: `auto`

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, and the end node and is expected to return elements (`(source-name, target-name, target-node) => elements`). If `auto` is given, a straight line will be drawn between nodes.

direction: `string` Default: `"down"`

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

parent-position: `string` Default: `"center"`

Positioning of parent nodes (begin, center, end)

grow: `float` Default: `1`

Depth grow factor

spread: `float` Default: 1

Sibling spread factor

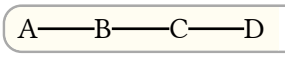
name: `none` or `string` Default: `none`

The tree elements name

Nodes

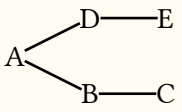
A tree node is an array consisting of the nodes value at index 0 followed by its child nodes. For the default `draw-node` function, the value (first item) of an node must be of type `content`.

Example of a list of nodes:




```
cetz.tree.tree(([A], ([B], ([C], ([D],)))), direction: "right")
```

Example of a tree of nodes:



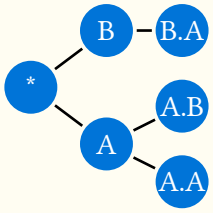

```
cetz.tree.tree(([A], ([B], [C]), ([D], [E])), direction: "right")
```

Drawing and Styling Tree Nodes

The `tree()` function takes an optional `draw-node:` and `draw-edge:` callback function that can be used to customice node and edge drawing.

The `draw-node` function must take the current node and its parents node anchor as arguments and return one or more elements.

For drawing edges between nodes, the `draw-edge` function must take two node anchors and the target node as arguments and return one or more elements.




```
import cetz.tree
set-style(content: (padding: .1))
let data = ([\*], ([A], [A.A], [A.B]), ([B], [B.A]))
tree.tree(
  data,
  direction: "right",
  draw-node: (node, ..) => {
    circle((), radius: .35, fill: blue, stroke: none)
    content((), text(white, [#node.content]))
  },
  draw-edge: (from, to, ..) => {
    let (a, b) = (from + ".center", to + ".center")
    line((a: a, b: b, abs: true, number: .40),
         (a: b, b: a, abs: true, number: .40))
  }
)
```

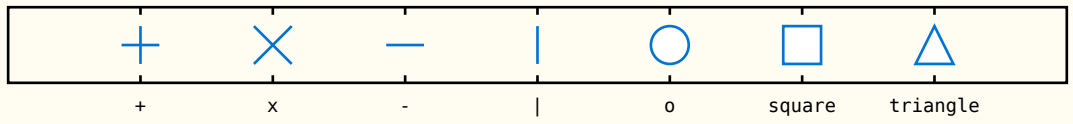
5.2 Plot

The library `plot` of CeTZ allows plotting data.

5.2.1 Types

Types commonly used by function of the `plot` library:

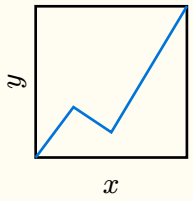
- `domain`: Tuple representing a functions domain as closed interval. Example domains are: `(0, 1)` for $[0, 1]$ or `(-calc.pi, calc.pi)` for $[-\pi, \pi]$.
- `axes`: Tuple of axis names. Plotting functions taking an `axes` tuple will use those axes as their `x` and `y` axis for plotting. To rotate a plot, you can simply swap its axes, for example `("y", "x")`.
- `mark`: Plots feature their own set of marks. The following mark symbols are available:




```
let marks = ("+", "x", "-", "|", "o", "square", "triangle")
cetz.plot.plot(size: (14, 1), x-min: 0, x-max: marks.len() + 1,
  x-ticks: marks.enumerate().map(((i, s)) => (i+1, raw(s))),
  x-tick-step: none, y-tick-step: none,
  x-label: none, y-label: none,
  {
  for (i, s) in marks.enumerate() {
    cetz.plot.add(((i + 1, 0),), mark: s, mark-style: (stroke: blue, fill: white), mark-size: .5)
  }
})
```

5.2.2 plot

Create a plot environment. Data to be plotted is given by passing it to the `plot.add` or other plotting functions. The plot environment supports different axis styles to draw, see its parameter `axis-style:`.




```
import cetz.plot
plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  plot.add(((0,0), (1,1), (2,.5), (4,3)))
})
```

To draw elements insides a plot, using the plots coordinate system, use the `plot.annotate(..)` function.

Parameters

plot(
  body: `body`,
  size: `array`,
  axis-style: `none` `string`,
  name: `string`,
  plot-style: `style` `function`,
  mark-style: `style` `function`,
  fill-below: `bool`,
  legend: `none` `auto` `coordinate`,
  legend-anchor: `auto` `string`,
  legend-style: `style`,
  ..options: `any`
)

body: `body`

Calls of `plot.add` or `plot.add-*` commands. Note that normal drawing commands like `line` or `rect` are not allowed inside the plots body, instead wrap them in `plot.annotate`, which lets you select the axes used for drawing.

size: `array` Default: `(1, 1)`

Plot size tuple of `(<width>, <height>)` in canvas units. This is the plots inner plotting size without axes and labels.

axis-style: `none` or `string` Default: `"scientific"`

How the axes should be styled:

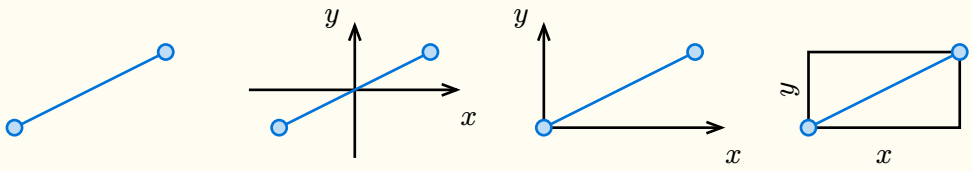
scientific Frames plot area using a rectangle and draw axes `x` (bottom), `y` (left), `x2` (top), and `y2` (right) around it. If `x2` or `y2` are unset, they mirror their opposing axis.

scientific-auto Draw set (used) axes `x` (bottom), `y` (left), `x2` (top) and `y2` (right) around the plotting area, forming a rect if all axes are in use or a L-shape if only `x` and `y` are in use.

school-book Draw axes `x` (horizontal) and `y` (vertical) as arrows pointing to the right/top with both crossing at $(0, 0)$

left Draw axes `x` and `y` as arrows, while the y axis stays on the left (at `x.min`) and the x axis at the bottom (at `y.min`)

`none` Draw no axes (and no ticks).




```
let opts = (x-tick-step: none, y-tick-step: none, size: (2,1))
let data = cetz.plot.add(((-1,-1), (1,1),), mark: "o")

for name in (none, "school-book", "left", "scientific") {
  cetz.plot.plot(axis-style: name, ..opts, data, name: "plot")
  content(((0,-1), "-|", "plot.south"), repr(name))
  set-origin((3.5,0))
}
```

name: `string` Default: `none`

The plots element name to be used when referring to anchors

plot-style: `style` or `function` Default: `default-plot-style`

Styling to use for drawing plot graphs. This style gets inherited by all plots and supports `palette` functions. The following style keys are supported:

stroke: `none` or `stroke` Default: `1pt`

Stroke style to use for stroking the graph.

fill: `none` or `paint` Default: `none`

Paint to use for filled graphs. Note that not all graphs may support filling and that you may have to enable filling per graph, see `plot.add(fill: ..)`.

mark-style: `style` or `function` Default: `default-mark-style`

Styling to use for drawing plot marks. This style gets inherited by all plots and supports `palette` functions. The following style keys are supported:

stroke: `none` or `stroke` Default: `1pt`

Stroke style to use for stroking the mark.

fill: `none` or `paint` Default: `none`

Paint to use for filling marks.

fill-below: `bool` Default: `true`

If true, the filled shape of plots is drawn *below* axes.

legend: `none` or `auto` or `coordinate` Default: `auto`

The position the legend will be drawn at. See Section 5.2.4 for information about legends. If set to `auto`, the legend's "default-placement" styling will be used. If set to a `coordinate`, it will be taken as relative to the plot's origin.

legend-anchor: `auto` or `string` Default: `auto`

Anchor of the legend group to use as its origin. If set to `auto` and `lengend` is one of the predefined legend anchors, the opposite anchor to `legend` gets used.

legend-style: `style` Default: `(:)`

Style key-value overwrites for the legend style with style root `legend`.

..options: `any`

Axis options, see *options* below.

Options

You can use the following options to customize each axis of the plot. You must pass them as named arguments prefixed by the axis name followed by a dash (`-`) they should target. Example: `x-min: 0`, `y-ticks: (..)` or `x2-label: [..]`.

label: `none` or `content` Default: `"none"`

The axis' label. If and where the label is drawn depends on the `axis-style`.

min: `auto` or `float` Default: `"auto"`

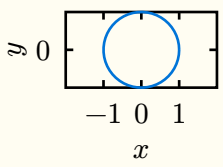
Axis lower domain value. If this is set greater than than `max`, the axis' direction is swapped

max: `auto` or `float` Default: `"auto"`

Axis upper domain value. If this is set to a lower value than `min`, the axis' direction is swapped

equal: `string` Default: `"none"`

Set the axis domain to keep a fixed aspect ratio by multiplying the other axis domain by the plots aspect ratio, depending on the other axis orientation (see `horizontal`). This can be useful to force one axis to grow or shrink with another one. You can only "lock" two axes of different orientations.




```
cetz.plot.plot(size: (2,1), x-tick-step: 1, y-tick-step: 1,
               x-equal: "y",
{
  cetz.plot.add(domain: (0, 2 * calc.pi),
    t => (calc.cos(t), calc.sin(t)))
})
```

horizontal: `bool` Default: `"axis name dependant"`

If true, the axis is considered an axis that gets drawn horizontally, vertically otherwise. The default value depends on the axis name on axis creation. Axes which name start with `x` have this set to `true`, all others have it set to `false`. Each plot has to use one horizontal and one vertical axis for plotting, a combination of two y-axes will panic: ("y", "y2").

tick-step: `none` or `auto` or `float` Default: `"auto"`

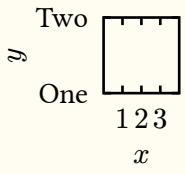
The increment between tick marks on the axis. If set to `auto`, an increment is determined. When set to `none`, incrementing tick marks are disabled.

minor-tick-step: `none` or `float` Default: `"none"`

Like `tick-step`, but for minor tick marks. In contrast to ticks, minor ticks do not have labels.

ticks: `none` or `array` Default: `"none"`

A List of custom tick marks to additionally draw along the axis. They can be passed as an array of `float` values or an array of `(<float>, <content>)` tuples for setting custom tick mark labels per mark.




```
cetz.plot.plot(x-tick-step: none, y-tick-step: none,
               x-min: 0, x-max: 4,
               x-ticks: (1, 2, 3),
               y-min: 1, y-max: 2,
               y-ticks: ((1, [One]), (2, [Two])),
{
  cetz.plot.add(((0,0),))
})
```

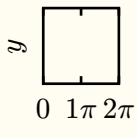
Examples: `(1, 2, 3)` or `((1, [One]), (2, [Two]), (3, [Three]))`

format: `none` or `string` or `function` Default: `"float"`

How to format the tick label: You can give a function that takes a `float` and return `content` to use as the tick label. You can also give one of the predefined options:

float Floating point formatting rounded to two digits after the point (see `decimals`)

sci Scientific formatting with $\times 10^n$ used as exponet syntax




```
let formatter(v) = if v != 0 {$ #{v/calc.pi} pi $} else {$ 0 $}
cetz.plot.plot(x-tick-step: calc.pi, y-tick-step: none,
               x-min: 0, x-max: 2 * calc.pi,
               x-format: formatter,
{
  cetz.plot.add(((0,0),))
})
```

decimals: `int` Default: `"2"`

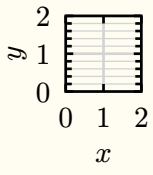
Number of decimals digits to display for tick labels, if the format is set to `"float"`.

unit: `none` or `content` Default: `"none"`

Suffix to append to all tick labels.

grid: `bool` or `string` Default: `"false"`

If `true` or `"major"`, show grid lines for all major ticks. If set to `"minor"`, show grid lines for minor ticks only. The value `"both"` enables grid lines for both, major- and minor ticks.

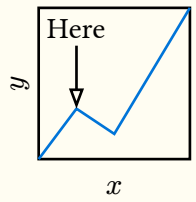



```
cetz.plot.plot(x-tick-step: 1, y-tick-step: 1,
               y-minor-tick-step: .2,
               x-min: 0, x-max: 2, x-grid: true,
               y-min: 0, y-max: 2, y-grid: "both", {
  cetz.plot.add(((0,0),))
})
```

5.2.3 add-anchor

Add an anchor to a plot environment

This function is similar to `draw.anchor` but it takes an additional axis tuple to specify which axis coordinate system to use.




```
import cetz.plot
import cetz.draw: *
plot.plot(size: (2,2), name: "plot",
          x-tick-step: none, y-tick-step: none, {
  plot.add(((0,0), (1,1), (2,.5), (4,3)))
  plot.add-anchor("pt", (1,1))
})

line("plot.pt", ((), "|-", (0,1.5)), mark: (start: ">"), name: "line")
content("line.end", [Here], anchor: "south", padding: .1)
```

Parameters

add-anchor(
  name: `string`,
  position: `tuple`,
  axes: `tuple`
)

name: `string`

Anchor name

position: `tuple`

Tuple of x and y values. Both values can have the special values “min” and “max”, which resolve to the axis min/max value. Position is in axis space defined by the axes passed to `axes`.

axes: `tuple` Default: `("x", "y")`

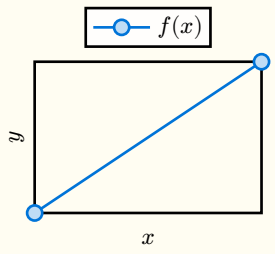
Name of the axes to use `("x", "y")` as coordinate system for `position`. Note that both axes must be used, as `add-anchors` does not create them on demand.

5.2.4 Legends

A legend for a plot will be drawn if at least one set of data with a label that is not `none` is given. The following anchors are available when placing a legend on a plot:

- `legend.north`
- `legend.south`
- `legend.east`
- `legend.west`
- `legend.north-east`
- `legend.north-west`
- `legend.south-east`
- `legend.south-west`
- `legend.inner-north`
- `legend.inner-south`
- `legend.inner-east`
- `legend.inner-west`
- `legend.inner-north-east`
- `legend.inner-north-west`
- `legend.inner-south-east`

- `legend.inner-south-west`




```
import cetz.plot
plot.plot(
  size: (3,2),
  x-tick-step: none,
  y-tick-step: none,
  legend: "legend.north", {
  plot.add(
    ((-1, -1), (1, 1)),
    mark: "o",
    label: $ f(x) $
  )
})
```

Styling

Root: `legend`

Keys

orientation: `direction` Default: `ttb`

The direction the legend items get laid out to.

default-position: `string` or `coordinate` Default: `"legend.north-east"`

The default position the legend gets placed at.

layer: `number` Default: `1`

The layer index the legend gets drawn at, see on-layer.

fill: `paint` Default: `rgb("#ffffffc8")`

The legends frame background color.

stroke: `stroke` Default: `luma(0%)`

The legends frame stroke style.

padding: `float` Default: `0.1`

The legends frame padding, that is the distance added between its items and its frame.

offset: `tuple` Default: `(0, 0)`

An offset tuple (x and y coordinates) to add to the legends position.

spacing: `number` Default: `0.1`

The spacing between the legend position and its frame.

item.spacing: `number` Default: `0.05`

The spacing between two legend items in canvas units.

item.preview.width: `number` Default: `0.75`

The width of a legend items preview picture, a small preview of the graph the legend item belongs to.

item.preview.height: `number` Default: `0.3`

The height of a legend items preview picture.

item.preview.margin: `number` Default: `0.1`

Margin between the preview picture and the item label.

5.2.5 add

Add data to a plot environment.

Note: You can use this for scatter plots by setting the stroke style to `none`: `add(..., style: (stroke: none))`.

Must be called from the body of a `plot(..)` command.

Parameters

```
add(
  domain: domain,
  hypograph: bool,
  epigraph: bool,
  fill: bool,
  fill-type: string,
  style: style,
  mark: string,
  mark-size: float,
  mark-style,
  samples: int,
  sample-at: array,
  line: string dictionary,
  axes: axes,
  label: none content,
  data: array function
)
```

domain: `domain` Default: `auto`

Domain of `data`, if `data` is a function. Has no effect if `data` is not a function.

hypograph: `bool` Default: `false`

Fill hypograph; uses the `hypograph` style key for drawing

epigraph: `bool` Default: `false`

Fill epigraph; uses the `epigraph` style key for drawing

fill: `bool` Default: `false`

Fill the shape of the plot

fill-type: `string` Default: `"axis"`

Fill type:

- **`"axis"`** Fill the shape to y = 0
- **`"shape"`** Fill the complete shape

style: `style` Default: `(:)`

Style to use, can be used with a `palette` function

mark: `string` Default: `none`

Mark symbol to place at each distinct value of the graph. Uses the `mark` style key of `style` for drawing.

mark-size: `float` Default: `.2`

Mark size in cavas units

mark-style: Default: `(:)`

samples: `int` Default: `50`

Number of times the `data` function gets called for sampling y-values. Only used if `data` is of type function. This parameter gets passed onto `sample-fn`.

sample-at: `array` Default: `()`

Array of x-values the function gets sampled at in addition to the default sampling. This parameter gets passed to `sample-fn`.

line: `string` or `dictionary` Default: `"linear"`

Line type to use. The following types are supported:

`"linear"` Draw linear lines between points

`"spline"` Calculate a Catmull-Rom through all points

`"vh"` Move vertical and then horizontal

`"hv"` Move horizontal and then vertical

`"hvh"` Add a vertical step in the middle

`"raw"` Like linear, but without linearization taking place. This is meant as a "fallback" for either bad performance or bugs.

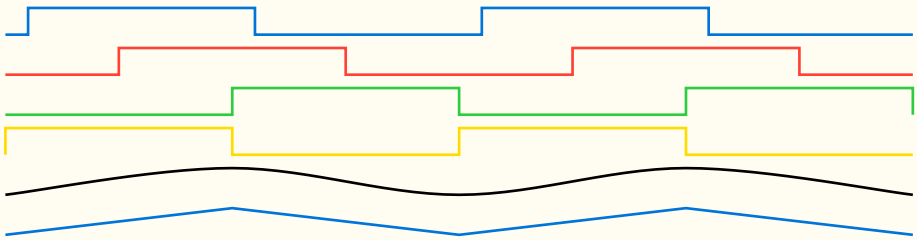
If the value is a dictionary, the type must be supplied via the `type` key. The following extra attributes are supported:

`"samples" <int>` Samples of splines

`"tension" <float>` Tension of splines

`"mid" <float>` Mid-Point of hvh lines (0 to 1)

`"epsilon" <float>` Linearization slope epsilon for use with `"linear"`, defaults to 0.



```
import cetz.plot
let points(offset: 0) = ((0,0), (1,1), (2,0), (3,1), (4,0)).map(((x,y)) => {
  (x,y + offset * 1.5)
})
plot.plot(size: (12, 3), axis-style: none, {
  plot.add(points(offset: 5), line: (type: "hvh", mid: .1))
  plot.add(points(offset: 4), line: "hvh")
  plot.add(points(offset: 3), line: "hv")
  plot.add(points(offset: 2), line: "vh")
  plot.add(points(offset: 1), line: "spline")
  plot.add(points(offset: 0), line: "linear")
})
```

axes: `axes` Default: `("x", "y")`

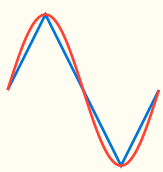
Name of the axes to use for plotting. Reversing the axes means rotating the plot by 90 degrees.

label: `none` or `content` Default: `none`

Legend label to show for this plot.

data: `array` or `function`

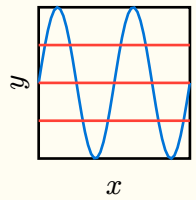
Array of 2D data points (numeric) or a function of the form `x => y`, where `x` is a value in `domain` and `y` must be numeric or a 2D vector (for parametric functions).



```
import cetz.plot
plot.plot(size: (2, 2), axis-style: none, {
  // Using an array of points:
  plot.add(((0,0), (calc.pi/2,1),
                   (1.5*calc.pi,-1), (2*calc.pi,0)))
  // Sampling a function:
  plot.add(domain: (0, 2*calc.pi), calc.sin)
})
```

5.2.6 add-hline

Add horizontal lines at one or more y-values. Every lines start and end points are at their axis bounds.




```
cetz.plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  cetz.plot.add(domain: (0, 4*calc.pi), calc.sin)
  // Add 3 horizontal lines
  cetz.plot.add-hline(-.5, 0, .5)
})
```

Parameters

add-hline(
  ..y: `float`,
  min: `auto` `float`,
  max: `auto` `float`,
  axes: `array`,
  style: `style`,
  label: `none` `content`
)

..y: `float`

Y axis value(s) to add a line at

min: `auto` or `float` Default: `auto`

X axis minimum value or auto to take the axis minimum

max: `auto` or `float` Default: `auto`

X axis maximum value or auto to take the axis maximum

axes: `array` Default: `("x", "y")`

Name of the axes to use for plotting

style: `style` Default: `(:)`

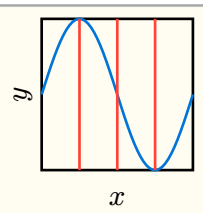
Style to use, can be used with a palette function

label: `none` or `content` Default: `none`

Legend label to show for this plot.

5.2.7 add-vline

Add vertical lines at one or more x-values. Every lines start and end points are at their axis bounds.




```
cetz.plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  cetz.plot.add(domain: (0, 2*calc.pi), calc.sin)
  // Add 3 vertical lines
  cetz.plot.add-vline(calc.pi/2, calc.pi, 3*calc.pi/2)
})
```

Parameters

```
add-vline(
  ..x: float,
  min: auto float,
  max: auto float,
  axes: array,
  style: style,
  label: none content
)
```

..x: `float`

X axis values to add a line at

min: `auto` or `float` Default: `auto`

Y axis minimum value or auto to take the axis minimum

max: `auto` or `float` Default: `auto`

Y axis maximum value or auto to take the axis maximum

axes: `array` Default: `("x", "y")`

Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!

style: `style` Default: `(:)`

Style to use, can be used with a palette function

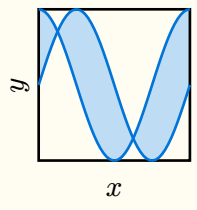
label: `none` or `content` Default: `none`

Legend label to show for this plot.

5.2.8 add-fill-between

Fill the area between two graphs. This behaves same as `add` but takes a pair of data instead of a single data array/function. The area between both function plots gets filled. For a more detailed explanation of the arguments, see `add()`.

This can be used to display an error-band of a function.




```
cetz.plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  cetz.plot.add-fill-between(domain: (0, 2*calc.pi),
    calc.sin, // First function/data
    calc.cos) // Second function/data
})
```

Parameters

```
add-fill-between(
  data-a: array function,
  data-b: array function,
  domain: domain,
  samples: int,
  sample-at: array,
  line: string dictionary,
  axes: array,
  label: none content,
  style: style
)
```

data-a: `array` or `function`

Data of the first plot, see `add()`.

data-b: `array` or `function`

Data of the second plot, see `add()`.

domain: `domain` Default: `auto`

Domain of both `data-a` and `data-b`. The domain is used for sampling functions only and has no effect on data arrays.

samples: `int` Default: `50`

Number of times the `data-a` and `data-b` function gets called for sampling y-values. Only used if `data-a` or `data-b` is of type function.

sample-at: `array` Default: `()`

Array of x-values the function(s) get sampled at in addition to the default sampling.

line: `string` or `dictionary` Default: `"linear"`

Line type to use, see `add()`.

axes: `array` Default: `("x", "y")`

Name of the axes to use for plotting.

label: `none` or `content` Default: `none`

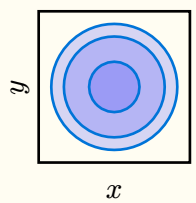
Legend label to show for this plot.

style: `style` Default: `(:)`

Style to use, can be used with a palette function.

5.2.9 add-contour

Add a contour plot of a sampled function or a matrix.




```
cetz.plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  cetz.plot.add-contour(x-domain: (-3, 3), y-domain: (-3, 3),
    style: (fill: rgb(50,50,250,50)),
    fill: true,
    op: "<",        // Find contours where data < z
    z: (2.5, 2, 1), // Z values to find contours for
    (x, y) => calc.sqrt(x * x + y * y))
})
```

Parameters

```
add-contour(
  data: array function,
  label: none content,
  z: float array,
  x-domain: domain,
  y-domain: domain,
  x-samples: int,
  y-samples: int,
  interpolate: bool,
  op: auto string function,
  axes: axes,
  style: style,
  fill: bool,
  limit: int
)
```

data: `array` or `function`

A function of the signature `(x, y) => z` or an array of arrays of floats (a matrix) where the first index is the row and the second index is the column.

label: `none` or `content` Default: `none`

Plot legend label to show. The legend preview for contour plots is a little rectangle drawn with the contours style.

z: `float` or `array` Default: `(1,)`

Z values to plot. Contours containing values above z (z >= 0) or below z (z < 0) get plotted. If you specify multiple z values, they get plotted in the order of specification.

x-domain: `domain` Default: `(0, 1)`

X axis domain used if `data` is a function, that is the domain inside the function gets sampled.

y-domain: `domain` Default: `(0, 1)`

Y axis domain used if `data` is a function, see `x-domain`.

x-samples: `int` Default: `25`

X axis domain samples (2 < n). Note that contour finding can be quite slow. Using a big sample count can improve accuracy but can also lead to bad compilation performance.

y-samples: `int` Default: `25`

Y axis domain samples (2 < n)

interpolate: `bool` Default: `true`

Use linear interpolation between sample values which can improve the resulting plot, especially if the contours are curved.

op: `auto` or `string` or `function` Default: `auto`

Z value comparison oparator:

`">", ">=", "<", "<=", "!=", "=="` Use the operator for comparison of `z` to the values from `data`.

`auto` Use ">=" for positive z values, "<=" for negative z values.

`function` Call comparison function of the format `(plot-z, data-z) => boolean`, where `plot-z` is the z-value from the plots `z` argument and `data-z` is the z-value of the data getting plotted. The function must return true if at the combinations of arguments a contour is detected.

axes: `axes` Default: `("x", "y")`

Name of the axes to use for plotting.

style: `style` Default: `(:)`

Style to use for plotting, can be used with a palette function. Note that all z-levels use the same style!

fill: `bool` Default: `false`

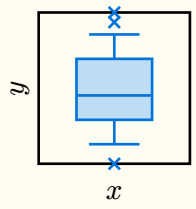
Fill each contour

limit: `int` Default: `50`

Limit of contours to create per z value before the function panics

5.2.10 add-boxwhisker

Add one or more box or whisker plots




```
cetz.plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  cetz.plot.add-boxwhisker((x: 1, // Location on x-axis
    outliers: (7, 65, 69),        // Optional outlier values
    min: 15, max: 60,             // Minimum and maximum
    q1: 25,                       // Quartiles: Lower
    q2: 35,                       //            Median
    q3: 50))                      //            Upper
})
```

Parameters

add-boxwhisker(
 data: `array` `dictionary`,
 label: `none` `content`,
 axes: `array`,
 style: `style`,
 box-width: `float`,
 whisker-width: `float`,
 mark: `string`,
 mark-size: `float`
)

data: `array` or `dictionary`

dictionary or array of dictionaries containing the needed entries to plot box and whisker plot.

The following fields are supported:

- `x` (number) X-axis value
- `min` (number) Minimum value
- `max` (number) Maximum value
- `q1`, `q2`, `q3` (number) Quartiles from lower to to upper
- `outliers` (array of number) Optional outliers

label: `none` or `content` Default: `none`

Legend label to show for this plot.

axes: `array` Default: `("x", "y")`

Name of the axes to use (“x”, “y”), note that not all plot styles are able to display a custom axis!

style: `style` Default: `(:)`

Style to use, can be used with a palette function

box-width: `float` Default: `0.75`

Width from edge-to-edge of the box of the box and whisker in plot units. Defaults to 0.75

whisker-width: `float` Default: `0.5`

Width from edge-to-edge of the whisker of the box and whisker in plot units. Defaults to 0.5

mark: `string` Default: `"*"`

Mark to use for plotting outliers. Set `none` to disable. Defaults to "x"

mark-size: `float` Default: `0.15`

Size of marks for plotting outliers. Defaults to 0.15

5.2.11 add-bar

Add a bar- or column-chart to the plot

A bar- or column-chart is a chart where values are drawn as rectangular boxes.

Parameters

```
add-bar(
  data: array,
  mode: string,
  labels: none content array,
  bar-width: float,
  bar-position: string,
  style: dictionary,
  axes: axes
)
```

data: `array`

Array of data items. An item is an array containing a x an one or more y values. For example `(0, 1)` or `(0, 10, 5, 30)`. Depending on the `mode`, the data items get drawn as either clustered or stacked rects.

mode: `string` Default: `"basic"`

The mode on how to group data items into bars:

basic Add one bar per data value. If the data contains multiple values, group those bars next to each other.

clustered Like "basic", but take into account the maximum number of values of all items and group each cluster of bars together having the width of the widest cluster.

stacked Stack bars of subsequent item values onto the previous bar, generating bars with the height of the sume of all an items values.

stacked100 Like "stacked", but scale each bar to height 100, making the different bars percentages of the sum of an items values.

labels: `none` or `content` or `array` Default: `none`

A single legend label for "basic" bar-charts, or a a list of legend labels per bar category, if the mode is one of "clustered", "stacked" or "stacked100".

bar-width: `float` Default: `1`

Width of one data item on the y axis

bar-position: `string` Default: `"center"`

Positioning of data items relative to their x value.

- "start": The lower edge of the data item is on the x value (left aligned)

- “center”: The data item is centered on the x value
- “end”: The upper edge of the data item is on the x value (right aligned)

style: `dictionary` Default: `(:)`

Plot style

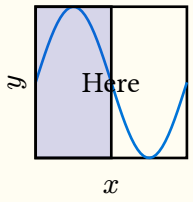
axes: `axes` Default: `("x", "y")`

Plot axes. To draw a horizontal growing bar chart, you can swap the x and y axes.

5.2.12 annotate

Add an annotation to the plot

An annotation is a sub-canvas that uses the plots coordinates specified by its x and y axis.




```
import cetz.plot
plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  plot.add(domain: (0, 2*calc.pi), calc.sin)
  plot.annotate({
    rect((0, -1), (calc.pi, 1), fill: rgb(50,50,200,50))
    content((calc.pi, 0), [Here])
  })
})
```

Bounds calculation is done naively, therefore fixed size content *can* grow out of the plot. You can adjust the padding manually to adjust for that. The feature of solving the correct bounds for fixed size elements might be added in the future.

Parameters

```
annotate(
  body: drawable,
  axes: axes,
  resize: bool,
  padding: none number dictionary,
  background: bool
)
```

body: `drawable`

Elements to draw

axes: `axes` Default: `("x", "y")`

X and Y axis names

resize: `bool` Default: `true`

If true, the plots axes get adjusted to contain the annotation

padding: `none` or `number` or `dictionary` Default: `none`

Annotation padding that is used for axis adjustment

background: `bool` Default: `false`

If true, the annotation is drawn behind all plots, in the background. If false, the annotation is drawn above all plots.

5.2.13 sample-fn

Sample the given single parameter function `samples` times, with values evenly spaced within the range given by `domain` and return each sampled `y` value in an array as `(x, y)` tuple.

If the functions first return value is a tuple `(x, y)`, then all return values must be a tuple.

Parameters

```
sample-fn(
  fn: function,
  domain: domain,
  samples: int,
  sample-at: array
) -> array: Array of (x y) tuples
```

fn: `function`

Function to sample of the form `(x) => y` or `(t) => (x, y)`, where `x` or `t` are `float` values within the domain specified by `domain`.

domain: `domain`

Domain of `fn` used as bounding interval for the sampling points.

samples: `int`

Number of samples in domain.

sample-at: `array` Default: `()`

List of x values the function gets sampled at in addition to the `samples` number of samples. Values outsides the specified domain are legal.

5.2.14 sample-fn2

Samples the given two parameter function with `x-samples` and `y-samples` values evenly spaced within the range given by `x-domain` and `y-domain` and returns each sampled output in an array.

Parameters

```
sample-fn2(
  fn: function,
  x-domain: domain,
  y-domain: domain,
  x-samples: int,
  y-samples: int
) -> array: Array of z scalars
```

fn: `function`

Function of the form `(x, y) => z` with all values being numbers.

x-domain: `domain`

Domain used as bounding interval for sampling point's x values.

y-domain: `domain`

Domain used as bounding interval for sampling point's y values.

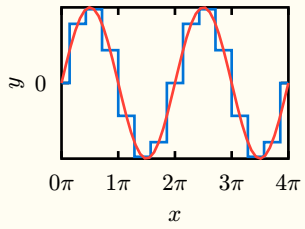
x-samples: `int`

Number of samples in the x-domain.

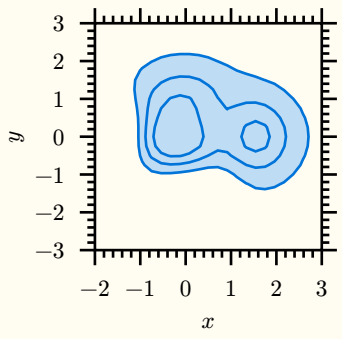
y-samples: `int`

Number of samples in the y-domain.

5.2.15 Examples




```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: calc.pi, y-tick-step: 1,
                       x-format: v => $#{v/calc.pi} pi$, {
  plot.add(domain: (0, 4*calc.pi), calc.sin,
    samples: 15, line: "hvh", style: (mark: (stroke: blue)))
  plot.add(domain: (0, 4*calc.pi), calc.sin)
})
```




```
import cetz.plot
import cetz.palette

// Let ticks point outwards by giving them negative length
set-style(axes: (tick: (length: -.2, minor-length: -.1)))

// Plot something
plot.plot(size: (3,3), x-tick-step: 1, x-minor-tick-step: .2,
                       y-tick-step: 1, y-minor-tick-step: .2, {
  let z(x, y) = {
    (1 - x/2 + calc.pow(x,5) + calc.pow(y,3)) * calc.exp(-(x*x) - (y*y))
  }
  plot.add-contour(x-domain: (-2, 3), y-domain: (-3, 3),
                   z, z: (.1, .4, .7), fill: true)
})
```

5.2.16 Styling

The following style keys can be used (in addition to the standard keys) to style plot axes. Individual axes can be styled differently by using their axis name as key below the `axes` root.

```
set-style(axes: ( /* Style for all axes */ ))
set-style(axes: (bottom: ( /* Style axis "bottom" */)))
```

Axis names to be used for styling:

- School-Book and Left style:
  - `x`: X-Axis
  - `y`: Y-Axis
- Scientific style:
  - `left`: Y-Axis
  - `right`: Y2-Axis
  - `bottom`: X-Axis
  - `top`: X2-Axis

Default `scientific` Style

```
(
  tick-limit: 100,
  minor-tick-limit: 1000,
  auto-tick-factors: (1, 1.5, 2, 2.5, 3, 4, 5, 6, 8, 10),
  auto-tick-count: 11,
  fill: none,
  stroke: luma(0%),
  label: (offset: 5.67pt, anchor: auto),
  tick: (
    fill: none,
    stroke: luma(0%),
    length: 2.83pt,
    minor-length: 80%,
    label: (offset: 5.67pt, angle: 0deg, anchor: auto),
  ),
```

```
  grid: (stroke: (paint: luma(83.33%), thickness: 1pt)),
  minor-grid: (stroke: (paint: luma(83.33%), thickness: 0.5pt)),
)
```

Default `school-book` Style

```
(
  tick-limit: 100,
  minor-tick-limit: 1000,
  auto-tick-factors: (1, 1.5, 2, 2.5, 3, 4, 5, 6, 8, 10),
  auto-tick-count: 11,
  fill: none,
  stroke: luma(0%),
  label: (offset: 5.67pt, anchor: auto),
  tick: (
    fill: none,
    stroke: luma(0%),
    length: 2.83pt,
    minor-length: 80%,
    label: (offset: 2.83pt, angle: 0deg, anchor: auto),
  ),
  grid: (stroke: (paint: luma(83.33%), thickness: 1pt)),
  minor-grid: (stroke: (paint: luma(83.33%), thickness: 0.5pt)),
  x: (stroke: auto, fill: none, mark: (end: "straight")),
  y: (stroke: auto, fill: none, mark: (end: "straight")),
  origin: (label: (offset: 1.42pt)),
  padding: 11.34pt,
)
```

5.3 Chart

With the `chart` library it is easy to draw charts.

5.3.1 barchart

Draw a bar chart. A bar chart is a chart that represents data with rectangular bars that grow from left to right, proportional to the values they represent. For examples see Section 5.3.5.

Styling

Root: `barchart`.

bar-width: `float` Default: `0.8`

Width of a single bar (basic) or a cluster of bars (clustered) in the plot.

y-inset: `float` Default: `1`

Distance of the plot data to the plot's edges on the y-axis of the plot.

You can use any `plot` or `axes` related style keys, too.

The `barchart` function is a wrapper of the `plot` API. Arguments passed to `..plot-args` are passed to the `plot.plot` function.

Parameters

```
barchart(
  data: array,
  label-key: int string,
  value-key: int string,
  mode: string,
  size: array,
  bar-style: style function,
  x-label: content none,
  x-unit: content auto,
  y-label: content none,
  labels: none content,
  ..plot-args: any
)
```

data: `array`

Array of data rows. A row can be of type array or dictionary, with `label-key` and `value-key` being the keys to access a rows label and value(s).

Example

```
(([A], 1), ([B], 2), ([C], 3),)
```

label-key: `int` or `string` Default: `0`

Key to access the label of a data row. This key is used as argument to the rows `.at(..)` function.

value-key: `int` or `string` Default: `1`

Key(s) to access value(s) of data row. These keys are used as argument to the rows `.at(..)` function.

mode: `string` Default: `"basic"`

Chart mode:

basic Single bar per data row
clustered Group of bars per data row
stacked Stacked bars per data row
stacked100 Stacked bars per data row relative to the sum of the row

size: `array` Default: `(auto, 1)`

Chart size as width and height tuple in canvas unist; width can be set to `auto`.

bar-style: `style` or `function` Default: `palette.red`

Style or function (idx => style) to use for each bar, accepts a palette function.

x-label: `content` or `none` Default: `none`

x axis label

x-unit: `content` or `auto` Default: `auto`

Tick suffix added to each tick label

y-label: `content` or `none` Default: `none`

Y axis label

labels: `none` or `content` Default: `none`

Legend labels per x value group

..plot-args: `any`

Arguments to pass to `plot.plot`

5.3.2 columnchart

Draw a column chart. A column chart is a chart that represents data with rectangular bars that grow from bottom to top, proportional to the values they represent. For examples see Section 5.3.6.

Styling

Root: `columnchart`.

bar-width: `float` Default: `0.8`

Width of a single bar (basic) or a cluster of bars (clustered) in the plot.

x-inset: `float` Default: `1`

Distance of the plot data to the plot's edges on the x-axis of the plot.

You can use any `plot` or `axes` related style keys, too.

The `columnchart` function is a wrapper of the `plot` API. Arguments passed to `..plot-args` are passed to the `plot.plot` function.

Parameters

columnchart(
 data: `array`,
 label-key: `int` `string`,
 value-key: `int` `string`,
 mode: `string`,
 size: `array`,
 bar-style: `style` `function`,
 x-label: `content` `none`,
 y-unit: `content` `auto`,
 y-label: `content` `none`,
 labels: `none` `content`,
 ..plot-args: `any`
)

data: `array`

Array of data rows. A row can be of type array or dictionary, with `label-key` and `value-key` being the keys to access a rows label and value(s).

Example

```
(([A], 1), ([B], 2), ([C], 3),)
```

label-key: `int` or `string` Default: `0`

Key to access the label of a data row. This key is used as argument to the rows `.at(..)` function.

value-key: `int` or `string` Default: `1`

Key(s) to access value(s) of data row. These keys are used as argument to the rows `.at(..)` function.

mode: `string` Default: `"basic"`

Chart mode:

basic Single bar per data row

clustered Group of bars per data row

stacked Stacked bars per data row

stacked100 Stacked bars per data row relative to the sum of the row

size: `array` Default: `(auto, 1)`

Chart size as width and height tuple in canvas unist; width can be set to `auto`.

bar-style: `style` or `function` Default: `palette.red`

Style or function (idx => style) to use for each bar, accepts a palette function.

x-label: `content` or `none` Default: `none`

x axis label

y-unit: `content` or `auto` Default: `auto`

Tick suffix added to each tick label

y-label: `content` or `none` Default: `none`

Y axis label

labels: `none` or `content` Default: `none`

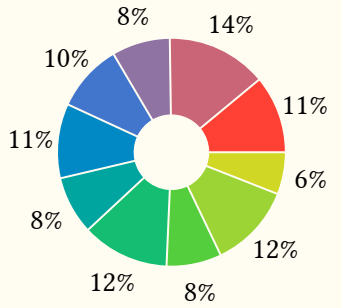
Legend labels per y value group

..plot-args: `any`

Arguments to pass to `plot.plot`

5.3.3 piechart

Draw a pie- or donut-chart




```
import cetz.chart
let data = (24, 31, 18, 21, 23, 18, 27, 17, 26, 13)
let colors = gradient.linear(red, blue, green, yellow)

chart.piechart(
  data,
  radius: 1.5,
  slice-style: colors,
  inner-radius: .5,
  outer-label: (content: "%",))
```

Styling

Root `piechart`

radius: `number` Default: `1`

Outer radius of the chart.

inner-radius: `number` Default: `0`

Inner radius of the chart slices. If greater than zero, the chart becomes a “donut-chart”.

gap: `number` or `angle` Default: `0.5deg`

Gap between chart slices to leave empty. This does not increase the charts radius by pushing slices outwards, but instead shrinks the slice. Big values can result in slices becoming invisible if no space is left.

outset-offset: `number` or `ratio` Default: `10%`

Absolute, or radius relative distance to push slices marked for “outsetting” outwards from the center of the chart.

outset-offset: `string` Default: `"OFFSET"`

The mode of how to perform “outsetting” of slices:

- "OFFSET": Offset slice position by `outset-offset`, increasing their gap to their siblings
- "RADIUS": Offset slice radius by `outset-offset`, which scales the slice and leaves the gap unchanged

start: `angle` Default: `0deg`

The pie-charts start angle. You can use this to draw charts not forming a full circle.

stop: `angle` Default: `360deg`

The pie-charts stop angle.

outer-label.content: `none` or `string` or `function` Default: `"LABEL"`

Content to display outsides the charts slices. There are the following predefined values:

LABEL Display the slices label (see `label-key`)

% Display the percentage of the items value in relation to the sum of all values, rounded to the next integer

VALUE Display the slices value

If passed a `function` of the format `(value, label) => content`, that function gets called with each slices value and label and must return content, that gets displayed.

outer-label.radius: `number` or `ratio` Default: `125%`

Absolute, or radius relative distance from the charts center to position outer labels at.

outer-label.angle: `angle` or `auto` Default: `0deg`

The angle of the outer label. If passed `auto`, the label gets rotated, so that the baseline is parallel to the slices secant.

outer-label.anchor: `string` Default: `"center"`

The anchor of the outer label to use for positioning.

inner-label.content: `none` or `string` or `function` Default: `none`

Content to display insides the charts slices. See `outer-label.content` for the possible values.

inner-label.radius: `number` or `ratio` Default: `150%`

Distance of the inner label to the charts center. If passed a `ratio`, that ratio is relative to the mid between the inner and outer radius (`inner-radius` and `radius`) of the chart

inner-label.angle: `angle` or `auto` Default: `0deg`

See `outer-label.angle`.

inner-label.anchor: `string` Default: `"center"`

See `outer-label.anchor`.

Anchors

The chart places one anchor per item at the radius of it's slice that gets named `"item-<index>"` (outer radius) and `"item-<index>-inner"` (inner radius), where index is the index of the sclice data in `data`.

Parameters

```
piechart(
  data: array,
  value-key: none int string,
  label-key: none int string,
  outset-key: none int string,
  outset: none int array,
  slice-style: function array gradient,
  name,
  ..style
)
```

data: `array`

Array of data items. A data item can be:

- A number: A number that is used as the fraction of the slice
- An array: An array which is read depending on value-key, label-key and outset-key
- A dictionary: A dictionary which is read depending on value-key, label-key and outset-key

value-key: `none` or `int` or `string` Default: `none`

Key of the "value" of a data item. If for example data items are passed as dictionaries, the value-key is the key of the dictionary to access the items chart value.

label-key: `none` or `int` or `string` Default: `none`

Same as the value-key but for getting an items label content.

outset-key: `none` or `int` or `string` Default: `none`

Same as the value-key but for getting if an item should get outset (highlighted). The outset can be a bool, float or ratio. If of type `bool`, the outset distance from the style gets used.

outset: `none` or `int` or `array` Default: `none`

A single or multiple indices of items that should get offset from the center to the outsides of the chart. Only used if outset-key is none!

slice-style: `function` or `array` or `gradient` Default: `palette.red`

Slice style of the following types:

- function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
- array: An array of style dictionaries or fill colors of at least one item. For each slice the style at the slices index modulo the arrays length gets used.
- gradient: A gradient that gets sampled for each data item using the the slices index divided by the number of slices as position on the gradient.

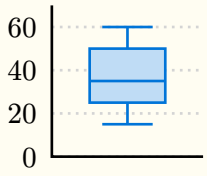
If one of stroke or fill is not in the style dictionary, it is taken from the charts style.

name: Default: `none`

..style:

5.3.4 boxwhisker

Add one or more box or whisker plots.




```
cetz.chart.boxwhisker(size: (2,2), label-key: none,
  y-min: 0, y-max: 70, y-tick-step: 20,
  (x: 1, min: 15, max: 60,
   q1: 25, q2: 35, q3: 50))
```

Styling

Root `boxwhisker`

box-width: `float` Default: `0.75`

The width of the box. Since boxes are placed 1 unit next to each other, a width of 1 would make neighbouring boxes touch.

whisker-width: `float` Default: `0.5`

The width of the whisker, that is the horizontal bar on the top and bottom of the box.

mark-size: `float` Default: `0.15`

The scaling of the mark for the boxes outlier values in canvas units.

You can use any `plot` or `axes` related style keys, too.

Parameters

boxwhisker(
 data: `array` `dictionary`,
 size,
 label-key: `integer` `string`,
 mark: `string`,
 ..plot-args: `any`
)

data: `array` or `dictionary`

Dictionary or array of dictionaries containing the needed entries to plot box and whisker plot.

See `plot.add-boxwhisker` for more details.

Examples:

- ```
  (x: 1                  // Location on x-axis
   outliers: (7, 65, 69), // Optional outliers
   min: 15, max: 60       // Minimum and maximum
   q1: 25,                // Quartiles: Lower
   q2: 35,                //            Median
   q3: 50)                //            Upper
  ```
- size (array) : Size of chart. If the second entry is auto, it automatically scales to accommodate the number of entries plotted

size: Default: `(1, auto)`

label-key: `integer` or `string` Default: `0`

Index in the array where labels of each entry is stored

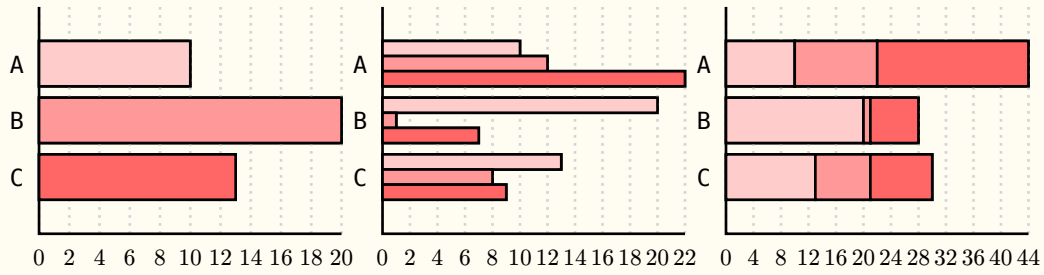
mark: `string` Default: `"*"`

Mark to use for plotting outliers. Set `none` to disable. Defaults to “x”

..plot-args: `any`
  ```

Additional arguments are passed to `plot.plot`

5.3.5 Examples – Bar Chart




```
import cetz.chart
// Left - Basic
let data = (("A", 10), ("B", 20), ("C", 13))
group(name: "a", {
  chart.barchart(size: (4, 3), data)
})
// Center - Clustered
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
group(name: "b", anchor: "south-west", {
  anchor("center", "a.south-east")
  chart.barchart(size: (4, 3), mode: "clustered", value-key: (1,2,3), data)
})
// Right - Stacked
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
group(name: "c", anchor: "south-west", {
  anchor("center", "b.south-east")
  chart.barchart(size: (4, 3), mode: "stacked", value-key: (1,2,3), data)
})
```

5.3.6 Examples – Column Chart

Basic, Clustered and Stacked


```
import cetz.chart
// Left - Basic
let data = (("A", 10), ("B", 20), ("C", 13))
group(name: "a", {
  chart.columnchart(size: (4, 3), data)
})
// Center - Clustered
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
group(name: "b", anchor: "south-west", {
  anchor("center", "a.south-east")
  chart.columnchart(size: (4, 3), mode: "clustered", value-key: (1,2,3), data)
})
// Right - Stacked
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
group(name: "c", anchor: "south-west", {
  anchor("center", "b.south-east")
  chart.columnchart(size: (4, 3), mode: "stacked", value-key: (1,2,3), data)
})
```

5.3.7 Styling

Charts share their axis system with plots and therefore can be styled the same way, see Section 5.2.16.

Default `barchart` Style

```
(
  axes: (
    tick: (length: 0),
    grid: (stroke: (dash: "dotted")),
  ),
  bar-width: 0.8,
  y-inset: 1,
)
```

Default `columnchart` Style

```
(
  axes: (
    tick: (length: 0),
    grid: (stroke: (dash: "dotted")),
  ),
  bar-width: 0.8,
  x-inset: 1,
)
```

Default `boxwhisker` Style

```
(
  axes: (
    tick: (length: 0),
    grid: (stroke: (dash: "dotted")),
  ),
  box-width: 0.75,
  whisker-width: 0.5,
  mark-size: 0.15,
)
```

Default `piechart` Style

```
(
  stroke: auto,
  fill: auto,
  radius: 1,
  inner-radius: 0,
  gap: 0.5deg,
  outset-offset: 10%,
  outset-mode: "OFFSET",
  start: 0deg,
  stop: 360deg,
  outer-label: (
    content: "LABEL",
    radius: 125%,
    angle: 0deg,
    anchor: "center",
  ),
  inner-label: (
    content: none,
    radius: 150%,
    angle: 0deg,
    anchor: "center",
```

```
  ),
)
```

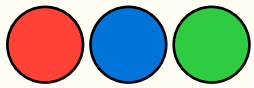
5.4 Palette

A palette is a function of the form `index => style` that takes an index, that can be any integer and returns a canvas style dictionary. If passed the string `"len"` it must return the length of its unique styles. An example use for palette functions is the `plot` library, which can use palettes to apply different styles per plot.

The palette library provides some predefined palettes.

5.4.1 new

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:

fill: `bool` Default: `true`

If true, the returned fill color is one of the colors from the `colors` list, otherwise the base styles fill is used.

stroke: `bool` Default: `false`

If true, the returned stroke color is one of the colors from the `colors` list, otherwise the base styles stroke color is used.

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

Parameters

```
new(
  base: style,
  colors: none array,
  dash: none array
)-> function Palette function of the form `index => style` that returns a style for an integer index
```

base: `style` Default: `base-style`

Style dictionary to use as base style for the styles generated per color

colors: `none` or `array` Default: `()`

List of colors the returned palette should return styles with

dash: `none` or `array` Default: `()`

List of stroke dash patterns the returned palette should return styles with

5.4.2 List of predefined palettes

- `gray`
- `red`
- `orange`
- `light-green`
- `dark-green`
- `turquoise`
- `cyan`
- `blue`
- `indigo`
- `purple`

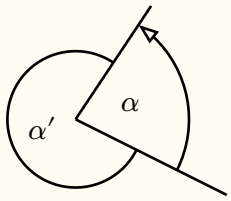
- `magenta`
- `pink`
- `rainbow`
- `tango-light`
- `tango`
- `tango-dark`

5.5 Angle

The `angle` function of the angle module allows drawing angles with an optional label.

5.5.1 angle

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




```
line((0,0), (1,1.5), name: "a")
line((0,0), (2,-1), 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: $ alpha' $,
  radius: 50%, inner: false)
```

Style Root: `angle`

Style Keys:

radius: `number` Default: `0.5`

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` or `ratio` Default: `50%`

The radius of the angles label origin. If of type `ratio`, it is relative to `radius`.

Anchors

`"a"` Point a
`"b"` Point b
`"origin"` Origin
`"label"` Label center
`"start"` Arc start
`"end"` Arc end

Parameters

angle(
 origin: `coordinate`,
 a: `coordinate`,
 b: `coordinate`,
 inner: `bool`,
 label: `none` `content` `function`,
 name: `none` `string`,
 ..style: `style`
)

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`.

inner: `bool` Default: `true`

Draw the smaller (inner) angle if true, otherwise the outer angle gets drawn.

label: `none` or `content` or `function` Default: `none`

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` or `string` Default: `none`

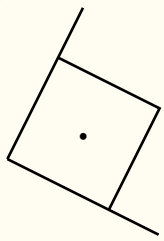
Element name, used for querying anchors.

..style: `style`

Style key-value pairs.

5.5.2 right-angle

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)
```

Style Root: `angle`

Style Keys:

radius: `number` Default: `0.5`

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` or `ratio` Default: `50%`

The radius of the angles label origin. If of type `ratio`, it is relative to the distance between `origin` and the angle corner.

Anchors

`"a"` Point a
`"b"` Point b
`"origin"` Origin
`"corner"` Angle corner
`"label"` Label center

Parameters

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

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` or `content` Default: `"•"`

Draw a label at the angles "label" anchor.

name: `none` or `string` Default: `none`

Element name, used for querying anchors.

..style: `style`

Style key-value pairs.

Default `angle` Style

```
(
  fill: none,
  stroke: auto,
  radius: 0.5,
  label-radius: 50%,
  mark: auto,
)
```

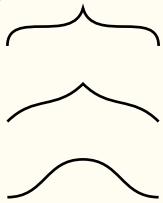
5.6 Decorations

Various pre-made shapes and path modifications.

5.6.1 Braces

5.6.2 brace

Draw a curly brace between two points.



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

cetz.decorations.brace((0,0),(2,0),
  pointiness: 45deg, outer-pointiness: 45deg)
cetz.decorations.brace((0,-1),(2,-1),
  pointiness: 90deg, outer-pointiness: 90deg)
```

Style Root: `brace`.
Style Keys:

amplitude: `number` Default: `0.5`

Sets the height of the brace, from its baseline to its middle tip.

pointiness: `ratio` or `angle` Default: `15deg`

How pointy the spike should be. `0deg` or `0%` for maximum pointiness, `90deg` or `100%` for minimum.

outer-pointiness: `ratio` or `angle` Default: `15deg`

How pointy the outer edges should be. `0deg` or `0` for maximum pointiness (allowing for a smooth transition to a straight line), `90deg` or `1` for minimum. Setting this to `auto` will use the value set for `pointiness`.

content-offset: `number` or `length` Default: `0.3`

Offset of the `"content"` anchor from the spike of the brace.

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

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

start: `coordinate`

Start point

end: `coordinate`

End point

flip: `bool` Default: `false`

Flip the brace around

debug: Default: `false`

name: `string` or `none` Default: `none`

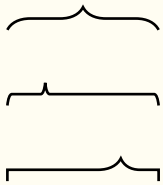
Element name used for querying anchors

..style: `style`

Style key-value pairs

5.6.3 flat-brace

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[1]. In contrast to `brace()`, these braces use straight line segments, resulting in better looks for long braces with a small amplitude.

Style Root: `flat-brace`
Style Keys:

amplitude: `number` Default: `0.3`

Determines how much the brace rises above the base line.

aspect: `ratio` Default: `50%`

Determines the fraction of the total length where the spike will be placed.

curves: `number` Default: `auto`

Curviness factor of the brace, a factor of 0 means no curves.

outer-curves: `auto` or `number` Default: `auto`

Curviness factor of the outer curves of the brace. A factor of 0 means no curves.

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

flat-brace(
  start: `coordinate`,
  end: `coordinate`,
  flip: `bool`,
  debug,
  name: `string` `none`,
  ..style: `style`
)

start: `coordinate`

Start point

end: `coordinate`

End point

flip: `bool` Default: `false`

Flip the brace around

[1] https://github.com/pgf-tikz/pgf/blob/6e5fd71581ab04351a89553a259b57988bc28140/tex/generic/pgf/libraries/decorations/pgflibrarydecorations.pathreplacing.code.tex#L136-L185

debug: Default: `false`

name: `string` or `none` Default: `none`

Element name for querying anchors

..style: `style`

Style key-value pairs

5.6.4 Path Decorations

Path decorations are elements that accept a path as input and generate one or more shapes that follow that path.

All path decoration functions support the following style keys:

start `ratio` or `length` (default: `0%`)

Absolute or relative start of the decoration on the path.

stop `ratio` or `length` (default: `100%`)

Absolute or relative end of the decoration on the path.

rest `string` (default: LINE)

If set to `"LINE"`, generate lines between the paths start/end and the decorations start/end if the path is *not closed*.

width `number` (default: 1)

Width or thickness of the decoration.

segments `int` (default: 10)

Number of repetitions/phases to generate. This key is ignored if `segment-length` is set != `none`.

segment-length `none` or `number`

Length of one repetition/phase of the decoration.

align `"START"`, `"MID"`, `"END"` (default: START)

Alignment of the decoration on the path *if `segment-length` is set* and the decoration does not fill up the full range between start and stop.

5.6.5 zigzag

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`

Keys

factor: `ratio` Default: `100%`

Triangle mid between its start and end. Setting this to 0% leads to a falling sawtooth shape, while 100% results in a raising sawtooth

Parameters

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

target: `drawable`

Target path

name: `none` or `string` Default: `none`

Element name

close: `auto` or `bool` Default: `auto`

Close the path

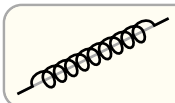
..style: `style`

Style

5.6.6 coil

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`

Keys

factor: `ratio` Default: `150%`

Factor of how much the coil overextends its length to form a curl.

Parameters

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

target: `drawable`

Target path

close: `auto` or `bool` Default: `auto`

Close the path

name: `none` or `string` Default: `none`

Element name

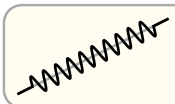
..style: `style`

Style

5.6.7 wave

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`

Keys

tension: `float` Default: `0.5`

Catmull-Rom curve tension, see `catmull()`.

Parameters

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

target: `drawable`

Target path

close: `auto` or `bool` Default: `auto`

Close the path

name: `none` or `string` Default: `none`

Element name

..style: `style`

Style

Styling

Default `brace` Style

```
(
  amplitude: 0.5,
  pointiness: 15deg,
  outer-pointiness: 0deg,
  content-offset: 0.3,
  debug-text-size: 6pt,
)
```

Default `flat-brace` Style

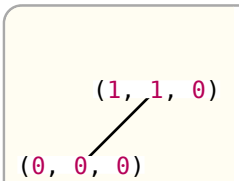
```
(
  amplitude: 0.3,
  aspect: 50%,
  curves: (1, 0.5, 0.6, 0.15),
  outer-curves: auto,
  content-offset: 0.3,
  debug-text-size: 6pt,
)
```

6 Advanced Functions

6.1 Coordinate

6.1.1 resolve

Resolve a list of coordinates to absolute 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

resolve(
  ctx: `context`,
  ..coordinates: `coordinate`,
  update: `bool`
) -> `(ctx vector..)` Returns a list of the new context object plus the

ctx: `context`

Canvas context object

..coordinates: `coordinate`

List of coordinates

update: `bool` Default: `true`

Update the context's last position resolved coordinate vectors

6.2 Styles

6.2.1 resolve

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")
)
#styles.resolve(dict, merge: (mark: (stroke: "yellow")), root: "line")
```

```
(
  stroke: "black",
  mark: (stroke: "yellow", fill: "blue"),
  fill: "red",
)
```

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: 1pt + luma(0%),
  fill: none,
  slant: none,
  harpoon: false,
  flip: false,
  reverse: false,
  flex: true,
  position-samples: 30,
  shorten-to: auto,
)
```

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

Parameters

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

dict: `style`

Current context style (`ctx.style`).

root: `none` or `str` Default: none

Style root element name.

merge: `style` Default: `(:)`

Style values overwriting the current style. I.e. inline styles passed with an element: `line(.., stroke: red)`.

base: `none` or `style` Default: `(:)`

Style values to merge into `dict` without overwriting it.

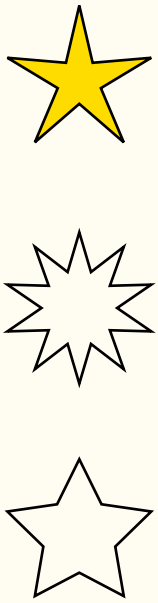
6.2.2 Default Style

This is a dump of the style dictionary every canvas gets initialized with. It contains all supported keys for all elements.

```
(
  fill: none,
  stroke: 1pt + luma(0%),
  radius: 1,
  shorten: "LINEAR",
  padding: none,
  mark: (
    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: auto,
    fill: auto,
    slant: none,
    harpoon: false,
    flip: false,
    reverse: false,
    flex: true,
    position-samples: 30,
    shorten-to: auto,
  ),
  circle: (radius: auto, stroke: auto, fill: auto),
  group: (padding: auto, fill: auto, stroke: auto),
  line: (mark: auto, fill: auto, stroke: auto),
  bezier: (
    stroke: auto,
    fill: auto,
    mark: auto,
    shorten: auto,
  ),
  catmull: (
    tension: 0.5,
    mark: auto,
    shorten: auto,
    stroke: auto,
    fill: auto,
  ),
  hobby: (
    omega: (1, 1),
    mark: auto,
    shorten: auto,
    stroke: auto,
    fill: auto,
  ),
  rect: (radius: 0, stroke: auto, fill: auto),
  arc: (
    mode: "OPEN",
    update-position: true,
    mark: auto,
    stroke: auto,
    fill: auto,
    radius: auto,
  ),
  content: (
    padding: auto,
    frame: none,
    fill: auto,
    stroke: auto,
  ),
)
```

7 Creating Custom Elements

The simplest way to create custom, reusable elements is to return them as a group. In this example we will implement a function `my-star(center)` that draws a star with `n` corners and a style specified inner and outer radius.



```
let my-star(center, name: none, ..style) = {
  group(name: name, ctx => {
    // Define a default style
    let def-style = (n: 5, inner-radius: .5, radius: 1)

    // Resolve the current style ("star")
    let style = cetz.styles.resolve(ctx.style, merge: style.named(),
      base: def-style, root: "star")

    // Compute the corner coordinates
    let corners = range(0, style.n * 2).map(i => {
      let a = 90deg + i * 360deg / (style.n * 2)
      let r = if calc.rem(i, 2) == 0 { style.radius } else { style.inner-radius }

      // Output a center relative coordinate
      (rel: (calc.cos(a) * r, calc.sin(a) * r, 0), to: center)
    })

    line(..corners, ..style, close: true)
  })
}

// Call the element
my-star((0,0))
my-star((0,3), n: 10)

set-style(star: (fill: yellow)) // set-style works, too!
my-star((0,6), inner-radius: .3)
```

8 Internals

8.1 Context

The state of the canvas is encoded in its context dictionary. Elements or other draw calls may return a modified context to the canvas to change its state, e.g. modifying the transformating matrix, adding a group or setting a style.

The context can be manually retrieved and modified using the `get-ctx` and `set-ctx` functions.

8.2 Elements

Each CeTZ element (`line`, `bezier`, `circle`, ...) returns an array of functions for drawing to the canvas. Such function takes the canvas' context and must return an dictionary of the following keys:

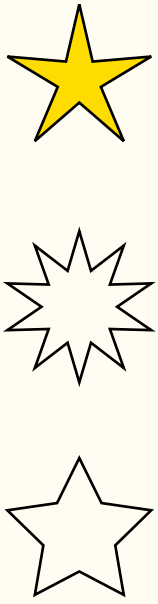
- `ctx` (required): The (modified) canvas context object
- `drawables`: List of drawables to render to the canvas
- `anchors`: A function of the form `(<anchor-identifier>) => <vector>`
- `name`: The elements name

An element that does only modify the context could be implemented like the following:

```
let my-element() = {
  (ctx => {
    // Do something with ctx ...
    (ctx: ctx)
  },)
}

// Call the element
my-element()
```

For drawing, elements must not use Typst native drawing functions, but output CeTZ paths. The `drawable` module provides functions for path creation (`path(..)`), the `path-util` module provides utilities for path segment creation. For demonstration, we will recreate the custmom element `my-star` from Section 7:



```
import cetz.drawable: path
import cetz.vector

let my-star(center, ..style) = {
  (ctx => {
    // Define a default style
    let def-style = (n: 5, inner-radius: .5, radius: 1, stroke: auto, fill: auto)

    // Resolve center to a vector
    let (ctx, center) = cetz.coordinate.resolve(ctx, center)

    // Resolve the current style ("star")
    let style = cetz.styles.resolve(ctx.style, merge: style.named(),
      base: def-style, root: "star")

    // Compute the corner coordinates
    let corners = range(0, style.n * 2).map(i => {
      let a = 90deg + i * 360deg / (style.n * 2)
      let r = if calc.rem(i, 2) == 0 { style.radius } else { style.inner-radius }
      vector.add(center, (calc.cos(a) * r, calc.sin(a) * r, 0))
    })

    // Build a path through all three coordinates
    let path = cetz.drawable.path((cetz.path-util.line-segment(corners),),
      stroke: style.stroke, fill: style.fill, close: true)

    (ctx: ctx,
     drawables: cetz.drawable.apply-transform(ctx.transform, path),
    )
  },)
}

// Call the element
my-star((0,0))
my-star((0,3), n: 10)
my-star((0,6), inner-radius: .3, fill: yellow)
```

Using custom elements instead of groups (as in Section 7) makes sense when doing advanced computations or even applying modifications to passed in elements.