

# The CeTZ package

Johannes Wolf and fenjalien  
<https://github.com/johannes-wolf/cetz>  
 Version 0.1.1

## Contents

1. Introduction .....	3
2. Usage .....	3
2.1. Argument Types .....	3
2.2. Anchors .....	3
3. Draw Function Reference .....	3
3.1. Canvas .....	3
3.2. Styling .....	4
3.3. Elements .....	5
3.3.1. line .....	5
3.3.2. rect .....	6
3.3.3. arc .....	7
3.3.4. circle .....	9
3.3.5. circle-through .....	10
3.3.6. bezier .....	11
3.3.7. bezier-through .....	12
3.3.8. content .....	13
3.3.9. catmull .....	14
3.3.10. grid .....	15
3.3.11. mark .....	16
3.4. Path Transformations .....	17
3.4.1. merge-path .....	17
3.4.2. group .....	18
3.4.3. anchor .....	19
3.4.4. copy-anchors .....	19
3.4.5. place-anchors .....	19
3.4.6. place-marks .....	20
3.4.7. intersections .....	21
3.5. Layers .....	21
3.5.1. on-layer .....	21
3.6. Transformations .....	22
3.6.1. translate .....	22
3.6.2. set-origin .....	23
3.6.3. set-viewport .....	23
3.6.4. rotate .....	24
3.6.5. scale .....	24
3.7. Context Modification .....	24
3.7.1. set-ctx .....	24
3.7.2. get-ctx .....	25
4. Coordinate Systems .....	25
4.1. XYZ .....	25
4.2. Previous .....	26

4.3. Relative .....	26
4.4. Polar .....	26
4.5. Barycentric .....	27
4.6. Anchor .....	27
4.7. Tangent .....	28
4.8. Perpendicular .....	28
4.9. Interpolation .....	29
4.10. Function .....	30
5. Utility .....	30
5.1.1. for-each-anchor .....	30
6. Libraries .....	31
6.1. Tree .....	31
6.1.1. tree .....	31
6.1.2. Node .....	33
6.2. Plot .....	33
6.2.1. add .....	33
6.2.2. add-anchor .....	36
6.2.3. plot .....	36
6.2.4. Examples .....	38
6.2.5. Styling .....	39
6.3. Chart .....	40
6.3.1. barchart .....	40
6.3.2. columnchart .....	42
6.3.3. Examples – Bar Chart .....	44
6.3.4. Examples – Column Chart .....	45
6.3.5. Styling .....	46
6.4. Palette .....	46
6.4.1. new .....	46
6.4.2. List of predefined palettes .....	47
6.5. Angle .....	47
6.5.1. angle .....	47
6.6. Decorations .....	49
6.6.1. brace .....	49
7. Advanced Functions .....	51
7.1. Coordinate .....	51
7.1.1. resolve .....	51
7.2. Styles .....	51
7.2.1. resolve .....	51

## 1. Introduction

This package provides a way to draw stuff 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”) and is pronounced like the word “Cats”.

## 2. Usage

This is the minimal starting point:

```
#import "@local/cetz:0.1.1"
#cetx.canvas({
  import cetx.draw: *
  ...
})
```

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

### 2.1. Argument Types

Argument types in this document are formatted in monospace and encased in angle brackets `<>`. Types such as `<integer>` and `<content>` are the same as Typst but additional are required:

**<coordinate>** Any coordinate system. See Section 4.  
**<number>** `<integer>` or `<float>`

### 2.2. Anchors

Anchors are named positions relative to named elements.

To use an anchor of an element, you must give the element a name using the `name` argument.



All elements will have default anchors based on their bounding box, they are: center, left, right, above/top and below/bottom, top-left, top-right, bottom-left, bottom-right. Some elements will have their own anchors.

Elements can be placed relative to their own anchors.



## 3. Draw Function Reference

### 3.1. Canvas

`canvas`(background: `none`, length: `1cm`, debug: `false`, body)

**background** <color> (default: none)

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

**length** <length> (default: 1cm)

Used to specify what 1 coordinate unit is.

**debug** <bool> (default: false)

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

**body**

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 have stroke and fill styling unless said otherwise.

**fill** <color> or <none> (default: none)

How to fill the draw element.

**stroke** <none> or <auto> or <length> (default: black + 1pt)

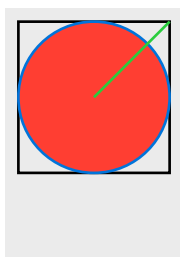
or <color> or <dictionary> or <stroke>

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.

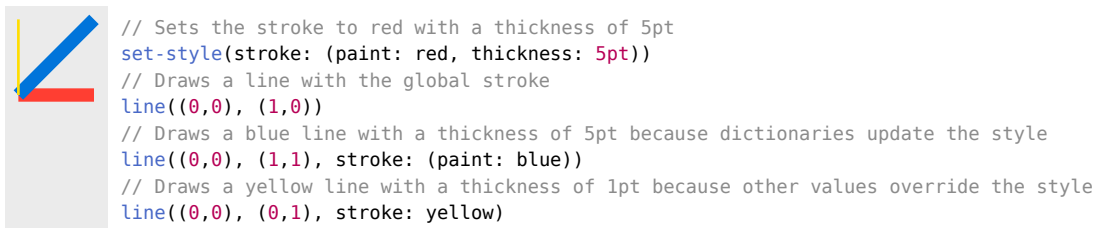


```
// Draws an empty square with a black border
rect((-1, -1), (1, 1))

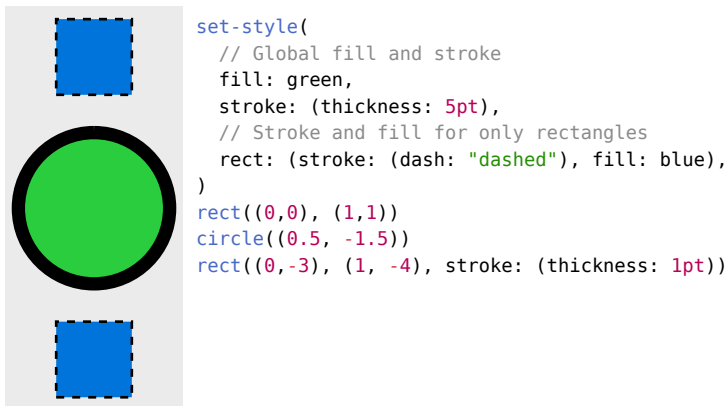
// Sets the global style to have a fill of red and a stroke of blue
set-style(stroke: blue, fill: red)
circle((0,0))

// Draws a green line despite the global stroke is blue
line((0,0), (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).



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.



### 3.3. Elements

#### 3.3.1. line

Draw a line or poly-line

Draws a line (a direct path between points) to the canvas. If multiple coordinates are given, a line is drawn between each consecutive one.

**Style root:** line.

**Anchor:**

- start – First coordinate
- end – Last coordinate

### 3.3.1.1. Parameters

```
line(
  ..pts-style: coordinate style,
  close: bool,
  name: string
)
```

**..pts-style** coordinate or style

- Coordinates to draw the line(s) between. A min. of two points must be given.
- Style attribute to set

**close** bool

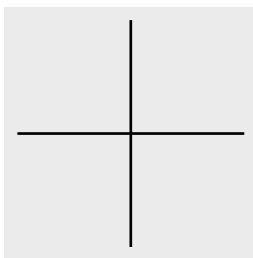
Close path. If true, a straight line is drawn from the last back to the first coordinate, closing the path.

Default: false

**name** string

Element name

Default: none



```
line((-1.5, 0), (1.5, 0))
line((0, -1.5), (0, 1.5))
```

### Styling

**mark** <dictionary> or <auto>

(default: auto)

The styling to apply to marks on the line, see mark

### 3.3.2. rect

Draw a rect from a to b

**Style root:** rect.

#### Anchors:

- center: Center
- top-left: Top left
- top-right: Top right
- bottom-left: Bottom left
- bottom-right: Bottom right
- top: Mid between top-left and top-right
- left: Mid between top-left and bottom-left
- right: Mid between top-right and bottom-right
- bottom: Mid between bottom-left and bottom-right

### 3.3.2.1. Parameters

```
rect(
  a: coordinate,
  b: coordinate,
  name: string,
  anchor: string,
  ..style: style
)
```

**a** coordinate

Bottom-Left coordinate

**b** coordinate

Top-Right coordinate

**name** string

Element name

Default: **none**

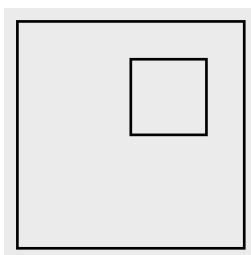
**anchor** string

Element origin

Default: **none**

**..style** style

Style



```
rect((0,0), (1,1))
rect((-1.5, 1.5), (1.5, -1.5))
```

### 3.3.3. arc

Draw an arc

**Style root:** arc.

Exactly two arguments of `start`, `stop` and `delta` must be set to a value other than `auto`. You can set the radius of the arc by setting the `radius` style option, which accepts a float or tuple of floats for setting the x/y radius. You can set the arcs draw mode using the `style mode`, which accepts the values "PIE", "CLOSE" and "OPEN" (default). If set to "PIE", the first and last points of the arc's path are it's center. If set to "CLOSE", the path is closed.

The arc curve is approximated using 1-4 cubic bezier curves.

### 3.3.3.1. Parameters

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

**position**    coordinate

Coordinate relative to anchor, which by default is the "start" anchor.

**start**    auto or angle

Start angle

Default: auto

**stop**    auto or angle

End angle

Default: auto

**delta**    auto or angle

Angle delta

Default: auto

**name**    none or string

Element name

Default: none

**anchor**    none or string

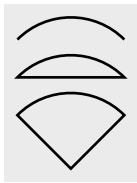
Anchor to position the arc at. If you want position the arc using its origin, pass the origin as position and set the anchor to "origin". Defaults to "start".

Default: none

**..style**    style

Style





```
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")
```

## Styling

**radius** <number> or <array> (default: 1)

The radius of the arc. This is also a global style shared with circle!

**mode** <string> (default: "OPEN")

The options are "OPEN" (the default, just the arc), "CLOSE" (a circular segment) and "PIE" (a circular sector).

### 3.3.4. circle

Draw a circle or an ellipse

**Style root:** circle.

The ellipses radii can be specified by its style field `radius`, which can be of type float or a tuple of two float's specifying the x/y radius.

#### 3.3.4.1. Parameters

```
circle(
  center: coordinate,
  name: string,
  anchor: string,
  ..style: style
)
```

**center**    coordinate

Center coordinate

**name**    string

Element name

Default: none

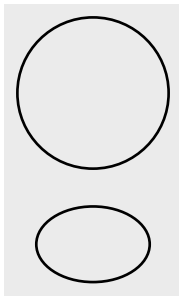
**anchor**    string

Element anchor

Default: none

**..style**    style

Style



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

### 3.3.5. circle-through

Draw a circle through three points

**Style root:** circle.

**Aliases:**

- a – Point a
- b – Point b
- c – Point c
- center – Calculated center

#### 3.3.5.1. Parameters

```
circle-through(
  a: coordinate,
  b: coordinate,
  c: coordinate,
  name: string,
  anchor: string,
  ..style
)
```

**a** coordinate

Point 1

**b** coordinate

Point 2

**c** coordinate

Point 3

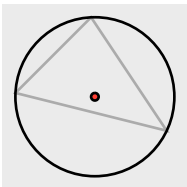
**name** string

Element name

Default: none

**anchor** `string`

Element name

Default: `none`**..style**

```
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)
```

## Styling

**radius** `<number>` or `<length>` or `<array of <number> or <length>>` (default: 1)

The circle's radius. If an array is given an ellipse will be drawn where the first item is the x radius and the second item is the y radius. This is also a global style shared with arc!

### 3.3.6. bezier

Draw a quadratic or cubic bezier line

**Style root:** bezier.

#### anchors:

- start – First coordinate
- end – Last coordinate
- ctrl-(n) – Control point (n)

#### 3.3.6.1. Parameters

```
bezier(
  start: coordinate,
  end: coordinate,
  ..ctrl-style: coordinate style,
  name: string
)
```

**start** `coordinate`

Start point

**end** `coordinate`

End point

**..ctrl-style** `coordinate` or `style`

Control points or Style attributes

**name** `string`

Element name

Default: `none`



```
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)
```

3.3.7. **bezier-through**

Draw a quadratic bezier from a to c through b

**Style root:** bezier.

3.3.7.1. **Parameters**

```
bezier-through(
  s: coordinate,
  b: coordinate,
  e: coordinate,
  name: string,
  ..style: style
)
```

**s** `coordinate`

Start point

**b** `coordinate`

Passthrough point

**e** `coordinate`

End point

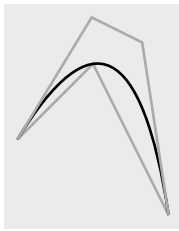
**name** `string`

Element name

Default: `none`

**..style** `style`

Style



```
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-1", "b.ctrl-2", c, stroke: gray)
```

### 3.3.8. content

Render content

**Style root:** content.

**Style keys:**

**padding (float)** Set vertical and horizontal padding

**frame (string, none)** Set frame style (none, "rect", "circle") The frame inherits the stroke and fill style.

NOTE: Content itself is not transformed by the canvas transformations! native transformation matrix support from typst would be required.

The following positional arguments are supported:

**coordinate, content** Place content at coordinate

**coordinate a, coordinate b, content** Place content in rect between a and b

#### 3.3.8.1. Parameters

```
content(
  angle: angle coordinate,
  clip: bool,
  anchor: string,
  name: string,
  ..style-args: coordinate content style
)
```

**angle** angle or coordinate

Rotation angle or coordinate relative to the first coordinate used for angle calculation

Default: 0deg

**clip** bool

Clip content inside rect

Default: false

**anchor** string

Anchor to use as origin. Defaults to "center" if one coordinate is set or "top-left" if two coordinates are set.

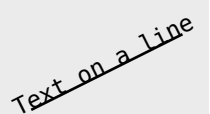
Default: none

**name** `string`

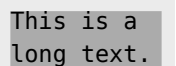
Node name

Default: `none`**..style-args** `coordinate` or `content` or `style`

Named arguments are used for for styling while positional args can be of coordinate or content, see the description above.


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


```
let (a, b) = ((1,0), (3,1))
line(a, b)
content((a, .5, b), angle: b, [Text on a line], anchor: "bottom")
```


`content((0,0), (2,1), par(justify: false)[This is a long text.], frame: "rect", fill: gray, stroke: none)`

## Styling

This draw element is not affected by fill or stroke styling.

**padding** `<length>` (default: `0pt`)

### 3.3.9. catmull

Draw a catmull-rom curve through points

**Style root:** `catmull`

#### 3.3.9.1. Parameters

```
catmull(
  ..points-style: coordinate style,
  tension: float,
  close: bool,
  name: string
)
```

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

List of points

**tension** `float`

Tension  $> .1$ , higher value means less curvature. The value is linear mapped  $t < .5 \rightarrow (0 < t < .5)$  and  $t > .5 \rightarrow (.5 < t < 10)$ .

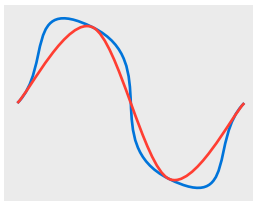
Default: `.5`

**close**    `bool`

Close the path

Default: `false`**name**    `string`

Element name

Default: `none`

```
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)
```

### 3.3.10. grid

Draw a grid

**Style root:** grid.

#### 3.3.10.1. Parameters

```
grid(
  from: coordinate,
  to: coordinate,
  step: float dictionary,
  name: string,
  help-lines: bool,
  ..style: style
)
```

**from**    `coordinate`

Start point

**to**    `coordinate`

End point

**step**    `float` or `dictionary`

Distance between grid lines. If passed a dictionary, *x* and *y* step can be set via the keys *x* and *y* ((*x*: <step>, *y*: <step>)).

Default: `1`

**name** `string`

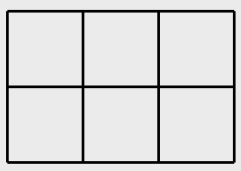
Element name

Default: `none`**help-lines** `bool`

Styles the grid using thin gray lines

Default: `false`**..style** `style`

Style

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

### 3.3.11. mark

Draw a mark or “arrow head” between two coordinates

**Style root:** mark.

Its styling influences marks being drawn on paths (line, bezier, ...).

#### 3.3.11.1. Parameters

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

**from** `coordinate`

Source coordinate

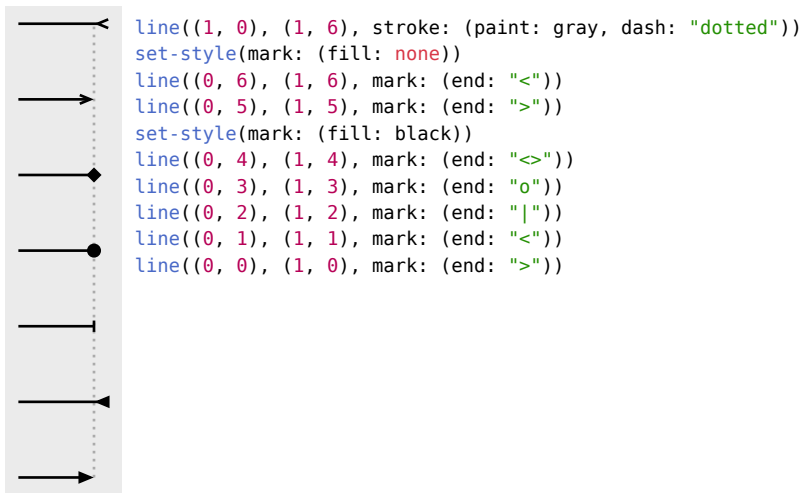
**to** `coordinate`

Target coordinate

**..style** `style`

Style





## Styling

**symbol** <string> (default: >)

The type of mark to draw when using the mark function.

**start** <string>

The type of mark to draw at the start of a path.

**end** <string>

The type of mark to draw at the end of a path.

**size** <number>

(default: 0.15)

The size of the marks.

**angle** <angle>

(default: 45deg)

Angle for triangle style marks (“<” and “>”)

## 3.4. Path Transformations

### 3.4.1. merge-path

Merge multiple paths

#### 3.4.1.1. Parameters

```

merge-path(
  body: any,
  close: bool,
  name: string,
  ..style
)

```

**body** any

Body

**close** bool

If true, the path is automatically closed

Default: false

**name** `string`

Element name

Default: `none`**..style**

```
// Merge two different paths into one
merge-path({
  line((0, 0), (1, 0))
  bezier((0, 0), (0, 0), (1,1), (0,1))
}, fill: white)
```

**3.4.2. group**

Push a group

A group has a local transformation matrix. Groups can be used to get an elements bounding box, as they set default anchors (top, top-left, ..) to the bounding box of their children.

Note: You can pass content a function of the form `ctx => draw-cmds` which returns the groups children. This way you get access to the groups context dictionary.

**3.4.2.1. Parameters**

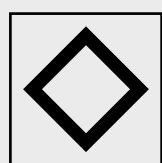
```
group(
  name: string,
  anchor: string,
  body: elements function
)
```

**name** `string`

Element name

Default: `none`**anchor** `string`

Element origin

Default: `none`**body** `elements` or `function`Children or function of the form `(ctx => elements)`

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

### 3.4.3. anchor

Register anchor name at position.

This only works inside a group!

#### 3.4.3.1. Parameters

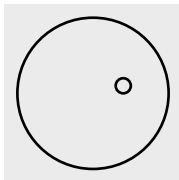
```
anchor(
  name: string,
  position: coordinate
)
```

**name** string

Anchor name

**position** coordinate

Coordinate



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

### 3.4.4. copy-anchors

Copy anchors of element to current group

#### 3.4.4.1. Parameters

```
copy-anchors(
  element: string,
  filter: none array
)
```

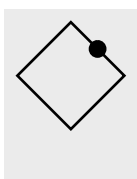
**element** string

Source element to copy anchors from

**filter** none or array

Name of anchors to copy or none to copy all

Default: none



```
group(name: "g", {
  rotate(45deg)
  rect((0,0), (1,1), name: "r")
  copy-anchors("r")
})
circle("g.top", radius: .1, fill: black)
```

### 3.4.5. place-anchors

Create anchors along a path

NOTE: This function is supposed to be replaced by a new coordinate syntax!

### 3.4.5.1. Parameters

```
place-anchors(
  path: path,
  ..anchors: positional,
  name: string
)
```

**path**    path

Path

**..anchors**    positional

List of dictionaries of the format: (name: string, pos: float), where pos is in range [0, 1].

**name**    string

Element name, uses paths name, if auto

Default: auto

```
place-anchors(name: "demo",
              bezier((0,0), (3,0), (1,-1), (2,1)),
                (name: "a", pos: .15),
                (name: "mid", pos: .5))
circle("demo.a", radius: .1, fill: black)
circle("demo.mid", radius: .1, fill: black)
```

### 3.4.6. place-marks

NOTE: This function is supposed to be removed!

Put marks on a path

#### 3.4.6.1. Parameters

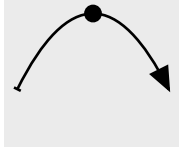
```
place-marks(
  path: path,
  ..marks-style: positional named,
  name
)
```

**path**    path

Path

**..marks-style**    positional or named

Array of dictionaries of the format: (mark: string, Mark symbol pos: float, Position between 0 and 1 name: string? Optional anchor name scale: float?, Optional scale angle: angle?, Optional angle (triangle marks only) stroke: stroke?, Optional stroke style fill: fill?) Optional fill style and style keys.

**name**Default: **none**

```
place-marks(bezier-through((0,0), (1,1), (2,0)),
  (mark: "|", size: .1, pos: 0),
  (mark: "o", size: .2, pos: .5),
  (mark: ">", size: .3, pos: 1),
  fill: black)
```

**3.4.7. intersections**

Emit one anchor per intersection of all elements inside body.

**3.4.7.1. Parameters**

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

**body** `elements`

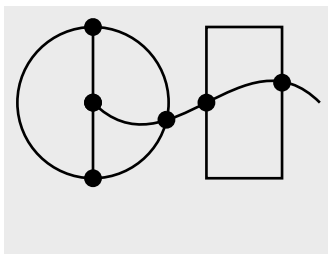
Element body

**name** `string`

Element name

Default: **none****samples** `int`

Number of samples to use for linearizing curves. Raising this gives more precision but slows down calculation.

Default: **10**

```
intersections(name: "demo", {
  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("demo", (name) => {
  circle("demo." + name, radius: .1, fill: black)
})
```

**3.5. Layers**

You can use layers to draw elements below or on top of other elements by using layers with a higher or lower index. When rendering, all draw commands are sorted by their layer (0 being the default).

**3.5.1. on-layer**

Draw body on layer

This can be used to draw elements behind already drawn elements by using a lower layer value (i.e -1). The layer value can be seen as a z-index.

### 3.5.1.1. Parameters

```
on-layer(
  layer: number,
  body: elements function
)
```

**layer**    number

Layer to draw the children at. The base layer is at 0 and all layers are drawn from low to high.

**body**    elements or function

Child elements or function (ctx => elements)



## 3.6. 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:

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

### 3.6.1. translate

Push translation matrix

#### 3.6.1.1. Parameters

```
translate(
  vec: vector dictionary,
  pre: bool
)
```

**vec**    vector or dictionary

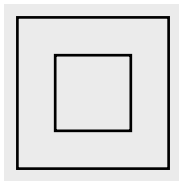
Translation vector

**pre**    bool

Specify matrix multiplication order

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

Default: true



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

### 3.6.2. set-origin

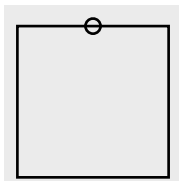
Sets the given position as the origin

#### 3.6.2.1. Parameters

`set-origin`(origin: coordinate)

**origin**    coordinate

Coordinate to set as new origin



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

### 3.6.3. set-viewport

Span rect between from and to as “viewport” with bounds bounds.

#### 3.6.3.1. Parameters

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

**from**    coordinate

Bottom-Left corner coordinate

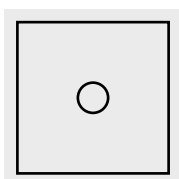
**to**    coordinate

Top right corner coordinate

**bounds**    vector

Bounds vector

Default: (1, 1, 1)



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

### 3.6.4. rotate

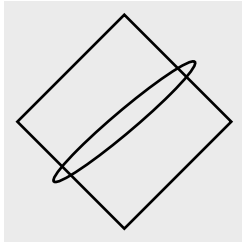
Rotate on z-axis (default) or specified axes if angle is of type dictionary.

#### 3.6.4.1. Parameters

`rotate`(angle: typst-angle dictionary)

**angle** typst-angle or dictionary

Angle (z-axis) or dictionary of the form (x: <typst-angle>, y: <angle>, z: <angle>) specifying per axis rotation typst-angle.



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

### 3.6.5. scale

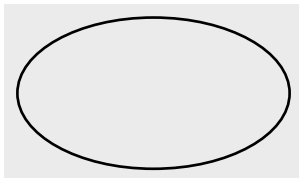
Push scale matrix

#### 3.6.5.1. Parameters

`scale`(factor: float dictionary)

**factor** float or dictionary

Scaling factor for all axes or per axis scaling factor dictionary.



```
// Scale x-axis
scale((x: 1.8))
circle((0,0))
```

## 3.7. Context Modification

The context of a canvas holds the canvas' internal state like style and transformation. Note that the fields of the context of a canvas are considered private and therefore unstable. You can add custom values to the context, but in order to prevent naming conflicts with future CeTZ versions, try to assign unique names.

### 3.7.1. set-ctx

Modify the canvas' context

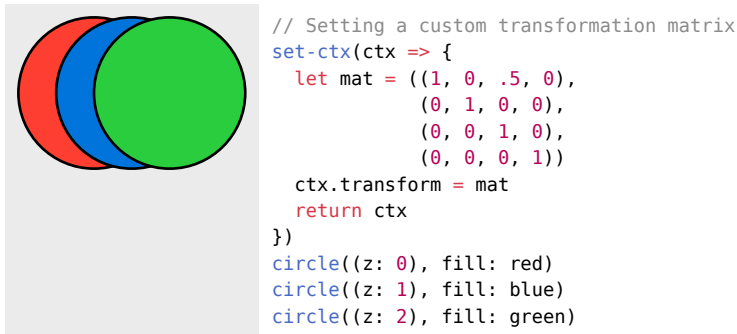
#### 3.7.1.1. Parameters

`set-ctx`(callback: function)

**callback** function

Function of the form `ctx => ctx` that returns the new canvas context.





### 3.7.2. get-ctx

Get the canvas' context and return children

#### 3.7.2.1. Parameters

`get-ctx`(body: function)

**body**      function

Function of the form `ctx => elements` that receives the current context and returns draw commands.

```
(
  (1.0, 0.0, 0.5, 0.0),
  (0.0, -1.0, -0.5, 0.0),
  (0, 0, 1, 0),
  (0, 0, 0, 1),
)

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

## 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> or <length> (default: 0)

The number of units in the *x* direction.

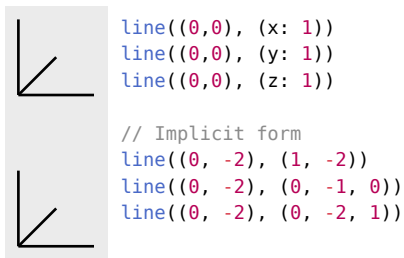
**y** <number> or <length> (default: 0)

The number of units in the *y* direction.

**z** <number> or <length> (default: 0)

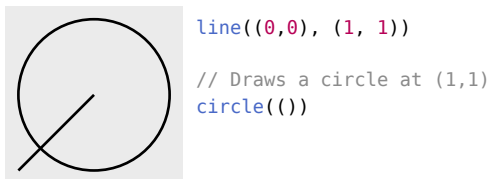
The number of units in the *z* direction.

The implicit form can be given as an array of two or three <number> or <length>, as in *(x,y)* and *(x,y,z)*.



## 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).



## 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** <bool>

(default: **true**)

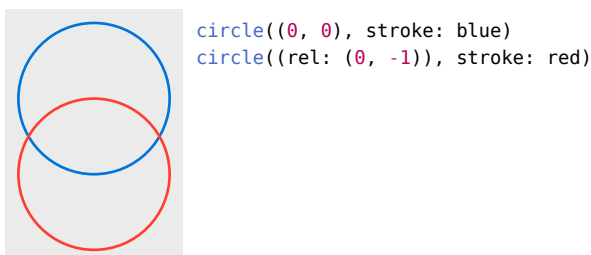
When false the previous position will not be updated.

**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.



## 4.4. Polar

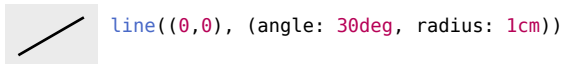
Defines a point a radius distance away from the origin at the given angle.

**angle** <angle>

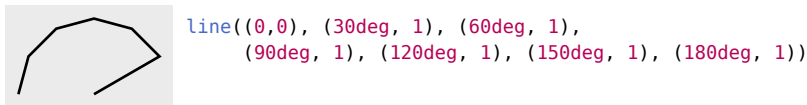
The angle of the coordinate. An angle of 0deg is to the right, a degree of 90deg is upward. See <https://typst.app/docs/reference/layout/angle/> for details.

**radius** <number> or <length> or <array of length or 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.



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



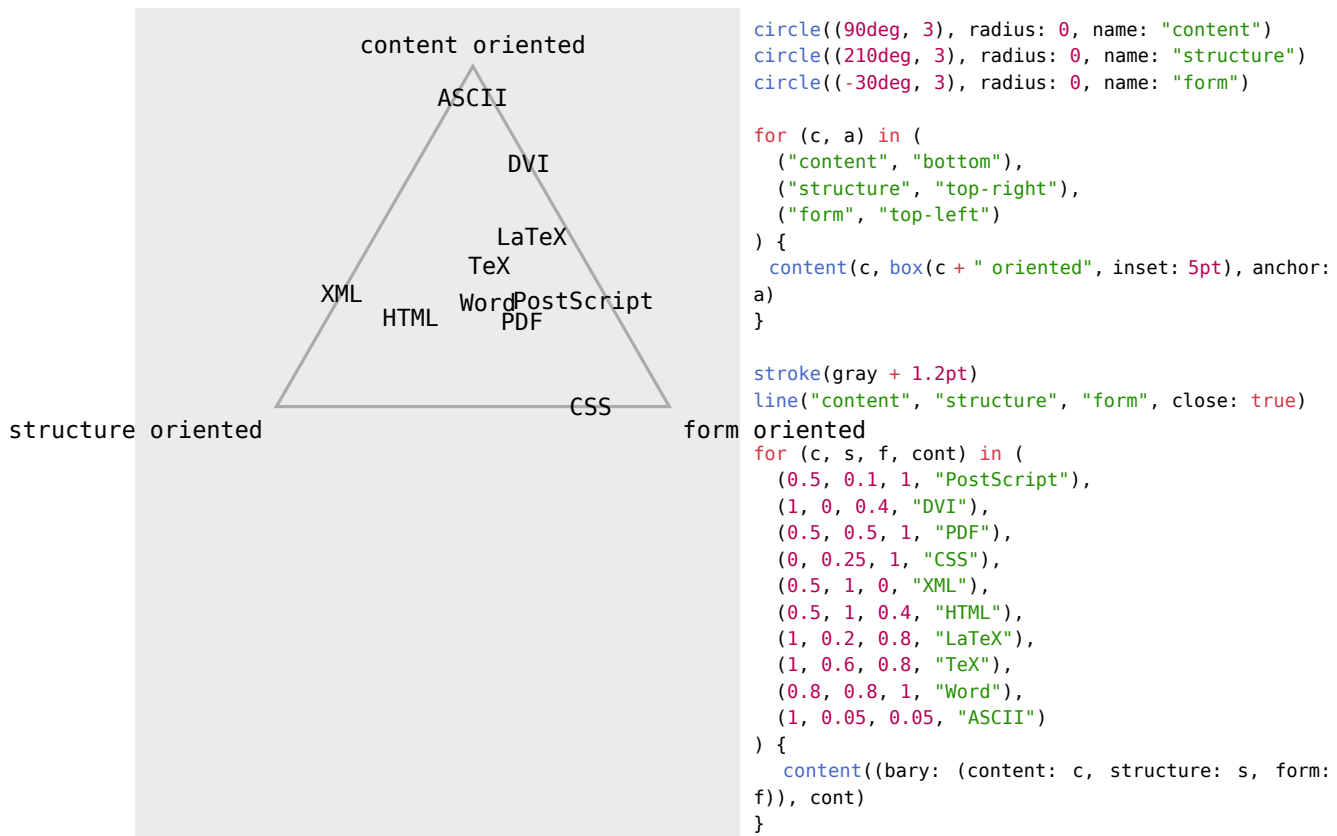
## 4.5. Barycentric

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors  $v_1, v_2, \dots, v_n$  and numbers  $\alpha_1, \alpha_2, \dots, \alpha_n$ . Then the barycentric coordinate specified by these vectors and numbers is

$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \dots + \alpha_n}$$

**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  $a$ .



## 4.6. Anchor

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

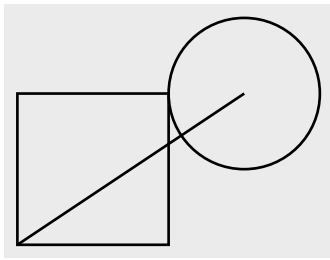
**name** <string>

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

**anchor** <string>

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".



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

## 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.

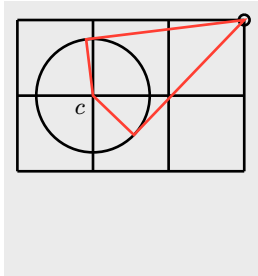
**point** <coordinate>

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: "top-right", 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  $q$ .

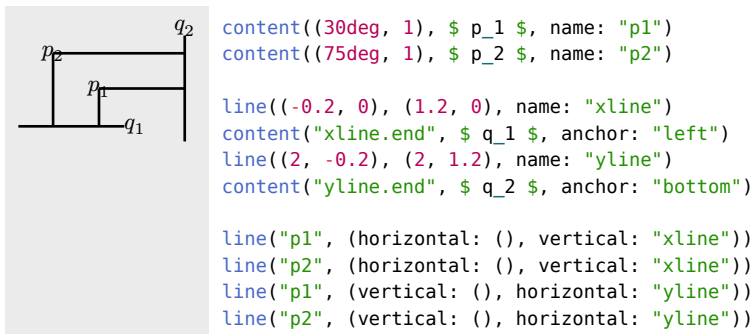
**horizontal** <coordinate>

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)



## 4.9. Interpolation

Use this to linearly interpolate between two coordinates *a* and *b* with a given factor number. If number is a <length> the position will be at the given distance away from *a* towards *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** <number> or <length>

The factor to interpolate by or the distance away from *a* towards *b*.

**angle** <angle>

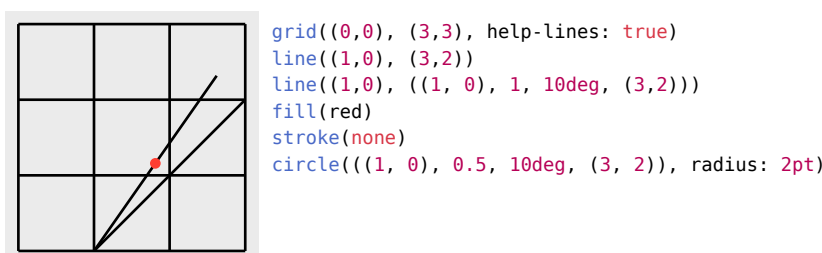
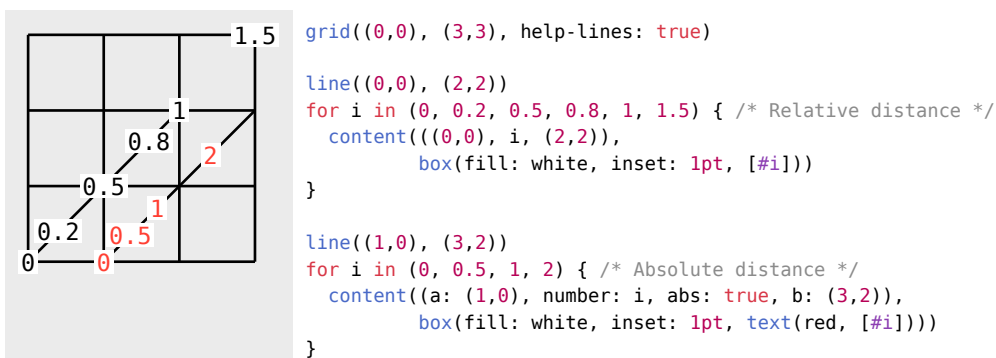
(default: 0deg)

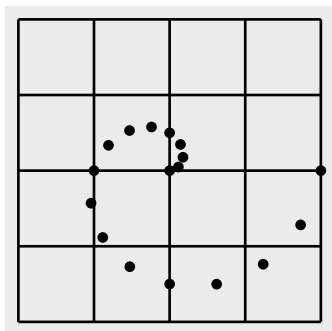
**abs** <bool>

(default: false)

Interpret number as absolute distance, instead of a factor.

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

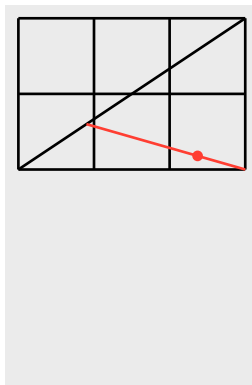




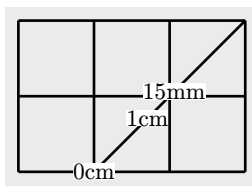
```
grid((0,0), (4,4), help-lines: true)

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), 0.3, (3, 2))),
    0.7,
    (3,0)
  ),
  radius: 2pt
)
```

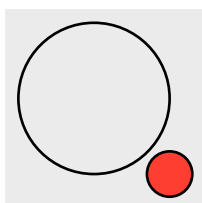


```
grid((0,0), (3, 2), help-lines: true)
line((1,0), (3,2))
for (l, c) in ((0cm, "0cm"), (1cm, "1cm"), (15mm, "15mm")) {
  content(((1,0), l, (3,2)), box(fill: white, $ #c $))
}
```

## 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.right"), radius: 0.3)
```

## 5. Utility

### 5.1.1. for-each-anchor

Execute callback for each anchor with the name of the anchor

The position of the anchor is set as the current position.

### 5.1.1.1. Parameters

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

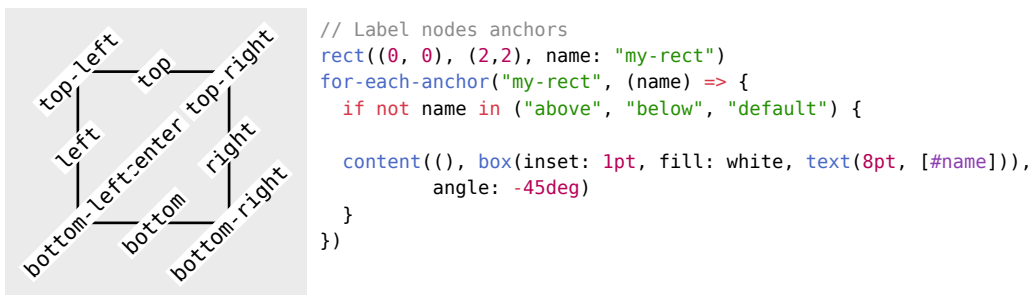
**node-prefix**    `string`

Anchor node name

**callback**    `function`

Callback of the form anchor-name => elements

Example: for-each-anchor("my-node", (name) => { content(), [#name] })



## 6. Libraries

### 6.1. Tree

With the tree library, CeTZ provides a simple tree layout algorithm.

- `tree()`

#### 6.1.1. tree

Layout and render tree nodes

##### 6.1.1.1. Parameters

```
tree(
  root: array,
  draw-node: function,
  draw-edge: function,
  direction: string,
  parent-position: string,
  grow: float,
  spread: float,
  name,
  ..style
)
```

**root**    `array`

Tree structure represented by nested lists Example: ([root], [child 1], ([child 2], [grandchild 1]))

**draw-node**    `function`

Callback for rendering a node. Signature: (node) => elements. The nodes position is accessible through the anchor "center" or the last position ().

Default: `auto`

**draw-edge**    `function`

Callback for rendering edges between nodes Signature: (source-name, target-name, target-node) => elements

Default: `auto`

**direction**    `string`

Tree grow direction (top, bottom, left, right)

Default: `"down"`

**parent-position**    `string`

Positioning of parent nodes (begin, center, end)

Default: `"center"`

**grow**    `float`

Depth grow factor (default 1)

Default: `1`

**spread**    `float`

Sibling spread factor (default 1)

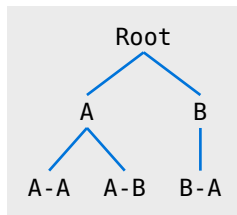
Default: `1`

**name**

Default: `none`

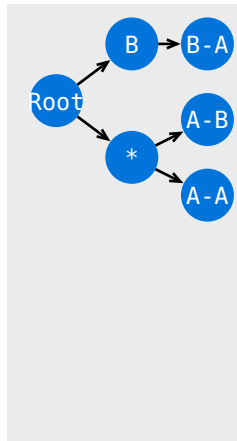
**..style**





```
import cetz.tree

let data = ([Root], ([A], [A-A], [A-B]), ([B], [B-A]))
tree.tree(data, content: (padding: .1), line: (stroke: blue))
```



```
import cetz.tree

let data = ([Root], ([\*], [A-A], [A-B]), ([B], [B-A]))
tree.tree(data, content: (padding: .1), direction: "right",
  mark: (end: ">", fill: none),
  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: .35),
      (a: b, b: a, abs: true, number: .35))
  })
```

### 6.1.2. Node

A tree node is an array of nodes. The first array item represents the current node, all following items are direct children of that node. The node itself can be of type content or dictionary with a key content.

## 6.2. Plot

The library plot of CeTZ allows plotting 2D data as linechart.

- `add()`
- `add-anchor()`
- `plot()`

### 6.2.1. add

Add data to a plot environment.

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

#### 6.2.1.1. Parameters

```
add(
  domain: array,
  hypograph: bool,
  epigraph: bool,
  fill: bool,
  style: style,
  mark: string,
  mark-size: float,
  mark-style,
  samples: int,
  sample-at: array,
  line: string dictionary,
  axes: array,
  data: array function
)
```

**domain**    array

Domain tuple of the plot. If data is a function, domain must be specified, as data is sampled for x-values in domain. Values must be numbers.

Default: **auto**

**hypograph**    bool

Fill hypograph; uses the hypograph style key for drawing

Default: **false**

**epigraph**    bool

Fill epigraph; uses the epigraph style key for drawing

Default: **false**

**fill**    bool

Fill to y zero

Default: **false**

**style**    style

Style to use, can be used with a palette function

Default: **(:)**

**mark**    string

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

The following marks are supported:

- "\*" or "x" – X
- "+" – Cross
- "|" – Bar
- "-" – Dash
- "o" – Circle
- "triangle" – Triangle
- "square" – Square

Default: **none**

**mark-size** `float`

Mark size in canvas units

Default: `.2`

**mark-style**

Default: `(:)`

**samples** `int`

Number of times the data function gets called for sampling y-values. Only used if data is of type function.

Default: `100`

**sample-at** `array`

Array of x-values the function gets sampled at in addition to the default sampling.

Default: `()`

**line** `string` or `dictionary`

Line type to use. The following types are supported:

- "linear"** Linear line segments
- "spline"** A smoothed line
- "vh"** Move vertical and then horizontal
- "hv"** Move horizontal and then vertical
- "vhv"** Add a vertical step in the middle
- "raw"** Like linear, but without linearization. *"linear"* *should* never look different than *"raw"*.

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 vhv lines (0 to 1)
- "epsilon"** `<float>` Linearization slope epsilon for use with *"linear"*, defaults to 0.

Default: `"linear"`

**axes** `array`

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

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

**data**    `array` or `function`

Array of 2D data points (numeric) or a function of the form  $x \Rightarrow y$ , where  $x$  is a value inside domain and  $y$  must be numeric or a 2D vector (for parametric functions).

#### Examples

- `((0,0), (1,1), (2,-1))`
- `x => calc.pow(x, 2)`

### 6.2.2. add-anchor

Add an anchor to a plot environment

#### 6.2.2.1. Parameters

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

**name**    `string`

Anchor name

**position**    `array`

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!

**axes**    `array`

Name of the axes to use (“ $x$ ”, “ $y$ ”), note that both axes must exist!

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

### 6.2.3. plot

Create a plot environment

Note: Data for plotting must be passed via `plot.add(...)`

Note that different axis-styles can show different axes. The “school-book” and “left” style shows only axis “ $x$ ” and “ $y$ ”, while the “scientific” style can show “ $x_2$ ” and “ $y_2$ ”, if set (if unset, “ $x_2$ ” mirrors “ $x$ ” and “ $y_2$ ” mirrors “ $y$ ”). Other axes (e.G. “my-axis”) work, but no ticks or labels will be shown.

#### Options

The following options are supported per axis and must be prefixed by <axis-name>-, e.G. x-min: 0.

- label (content): Axis label
- min (int): Axis minimum value
- max (int): Axis maximum value
- tick-step (float): Distance between major ticks
- minor-tick-step (float): Distance between minor ticks
- ticks (array): List of ticks values or value/label tuples. Example (1,2,3) or ((1, [A]), (2, [B]),)
- format (string): Tick label format, "float", "sci" (scientific) or a custom function that receives a value and returns a content (value => content).
- grid (bool,string): Enable grid-lines at tick values:
  - "major": Enable major tick grid
  - "minor": Enable minor tick grid
  - "both": Enable major & minor tick grid
  - false: Disable grid
- unit (content): Tick label suffix
- decimals (int): Number of decimals digits to display for tick labels

#### 6.2.3.1. Parameters

```
plot(
  body: body,
  size: array,
  axis-style: string,
  name: string,
  plot-style: style function,
  mark-style: style function,
  ..options: any
)
```

**body**    body

Calls of `plot.add` commands

**size**    array

Plot canvas size tuple of width and height in canvas units

Default: (1, 1)

**axis-style**    string

Axis style "scientific", "left", "school-book"

- "scientific": Frame plot area and draw axes y, x, y2, and x2 around it
- "school-book": Draw axes x and y as arrows with both crossing at (0,0)
- "left": Draw axes x and y as arrows, the y axis stays on the left (at x.min) and the x axis at the bottom (at y.min)

Default: "scientific"

**name** string

Element name

Default: none

**plot-style** style or function

Style used for drawing plot graphs This style gets inherited by all plots.

Default: default-plot-style

**mark-style** style or function

Style used for drawing plot marks. This style gets inherited by all plots.

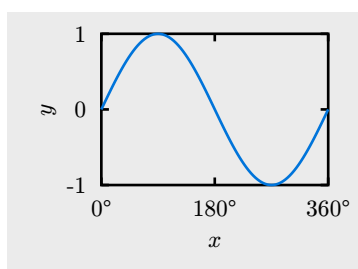
Default: default-mark-style

**..options** any

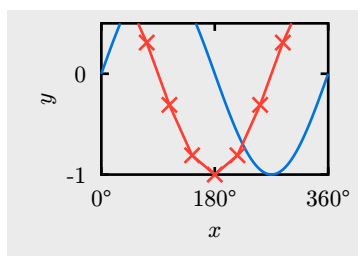
The following options are supported per axis and must be prefixed by <axis-name>-, e.G. x-min: 0.

- min (int): Axis minimum
- max (int): Axis maximum
- tick-step (float): Major tick step
- minor-tick-step (float): Major tick step
- ticks (array): List of ticks values or value/label tuples
- unit (content): Tick label suffix
- decimals (int): Number of decimals digits to display

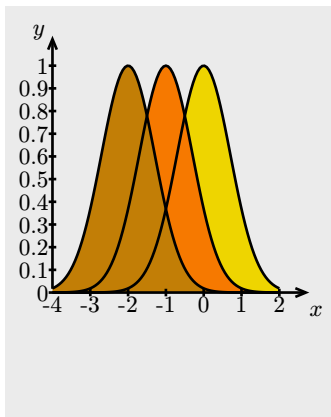
#### 6.2.4. Examples



```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 180, y-tick-step: 1,
          x-unit: $degree$, {
    plot.add(domain: (0, 360), x => calc.sin(x * 1deg))
  })
```



```
import cetz.plot
plot.plot(size: (3,2), x-tick-step: 180, y-tick-step: 1,
          x-unit: $degree$, y-max: .5, {
    plot.add(domain: (0, 360), x => calc.sin(x * 1deg))
    plot.add(domain: (0, 360), x => calc.cos(x * 1deg),
              samples: 10, mark: "x", style: (mark: (stroke: blue)))
  })
```



```
import cetz.plot
import cetz.palette

// Axes can be styled!
// Set the tick length to -.05:
set-style(axes: (tick: (length: -.05)))

// Plot something
plot.plot(size: (3,3), x-tick-step: 1, axis-style: "left", {
  for i in range(0, 3) {
    plot.add(domain: (-4, 2),
      x => calc.exp(-(calc.pow(x + i, 2))),
      fill: true, style: palette.tango)
  }
})
```

### 6.2.5. 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

#### 6.2.5.1. Default scientific Style

```
(
  fill: none,
  stroke: rgb("#000000"),
  label: (offset: 0.2),
  tick: (
    fill: none,
    stroke: rgb("#000000"),
    length: 0.1,
    minor-length: 0.08,
    label: (offset: 0.2, angle: 0deg, anchor: auto),
  ),
  grid: (
    stroke: (paint: rgb("#aaaaaa"), dash: "dotted"),
    fill: none,
  ),
)
```

#### 6.2.5.2. Default school-book Style

```
(
  fill: none,
  stroke: rgb("#000000"),
  label: (offset: 0.2),
  tick: (
    fill: none,
```

```

        stroke: rgb("#000000"),
        length: 0.1,
        minor-length: 0.08,
        label: (offset: 0.1, angle: 0deg, anchor: auto),
    ),
    grid: (
        stroke: (paint: rgb("#aaaaaa"), dash: "dotted"),
        fill: none,
    ),
    mark: (end: ">"),
    padding: 0.4,
)

```

## 6.3. Chart

With the chart library it is easy to draw charts.

Supported charts are:

- `barchart(. .)`: A chart with horizontal growing bars
  - `mode: "basic"`: (default): One bar per data row
  - `mode: "clustered"`: Multiple grouped bars per data row
  - `mode: "stacked"`: Multiple stacked bars per data row
  - `mode: "stacked100"`: Multiple stacked bars relative to the sum of a data row
- `barchart()`
- `columnchart()`

### 6.3.1. 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 6.3.3.

**Style root:** `barchart`.

#### 6.3.1.1. Parameters

```

barchart(
  data: array,
  label-key: int string,
  value-key: int string,
  mode: string,
  size: array,
  bar-width: float,
  bar-style: style function,
  x-tick-step: float,
  x-ticks: array,
  x-unit: content auto,
  x-label: content none,
  y-label: content none
)

```

**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`Key to access the label of a data row. This key is used as argument to the rows `.at( . )` function.Default: `0`**value-key** `int` or `string`Key(s) to access value(s) of data row. These keys are used as argument to the rows `.at( . )` function.Default: `1`**mode** `string`

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

Default: `"basic"`**size** `array`

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

Default: `(1, auto)`**bar-width** `float`

Size of a bar in relation to the charts height.

Default: `.8`**bar-style** `style` or `function`Style or function (`idx => style`) to use for each bar, accepts a palette function.Default: `palette.red`**x-tick-step** `float`

Step size of x axis ticks

Default: `auto`

**x-ticks**    `array`

List of tick values or value/label tuples

**Example**`(1, 5, 10) or ((1, [One]), (2, [Two]), (10, [Ten]))`Default: `()`**x-unit**    `content` or `auto`

Tick suffix added to each tick label

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

X axis label

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

Y axis label

Default: `none`

### 6.3.2. columnchart

Draw a column chart. A bar 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 6.3.4.

**Style root:** columnchart.

#### 6.3.2.1. Parameters

```
columnchart(
  data: array,
  label-key: int string,
  value-key: int string,
  mode: string,
  size: array,
  bar-width: float,
  bar-style: style function,
  x-label: content none,
  y-tick-step: float,
  y-ticks: array,
  y-unit: content auto,
  y-label: content none
)
```

**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`

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

Default: `0`

**value-key** `int` or `string`

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

Default: `1`

**mode** `string`

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

Default: `"basic"`

**size** `array`

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

Default: `(auto, 1)`

**bar-width** `float`

Size of a bar in relation to the charts height.

Default: `.8`

**bar-style** `style` or `function`

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

Default: `palette.red`

**x-label** `content` or `none`

x axis label

Default: `none`**y-tick-step** `float`

Step size of y axis ticks

Default: `auto`**y-ticks** `array`

List of tick values or value/label tuples

**Example**`(1, 5, 10)` or `((1, [One]), (2, [Two]), (10, [Ten]))`Default: `()`**y-unit** `content` or `auto`

Tick suffix added to each tick label

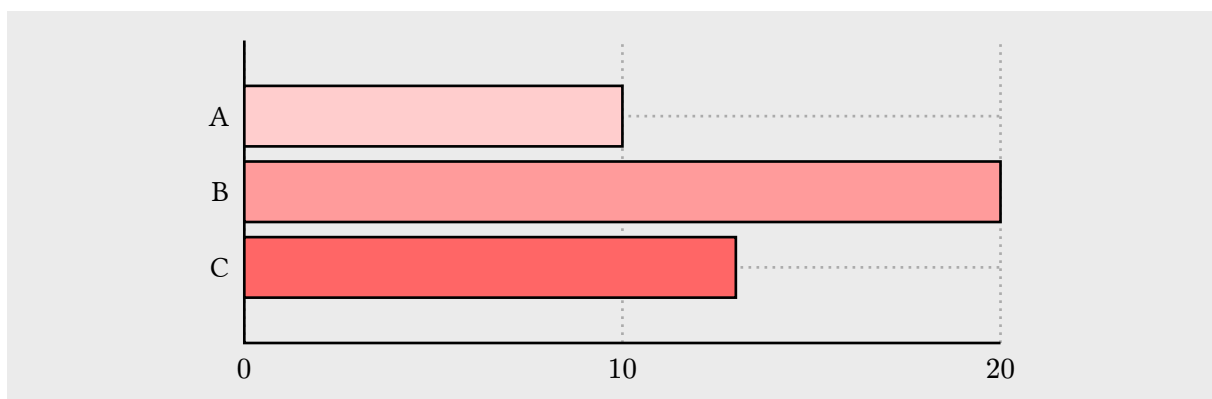
Default: `auto`**y-label** `content` or `none`

Y axis label

Default: `none`

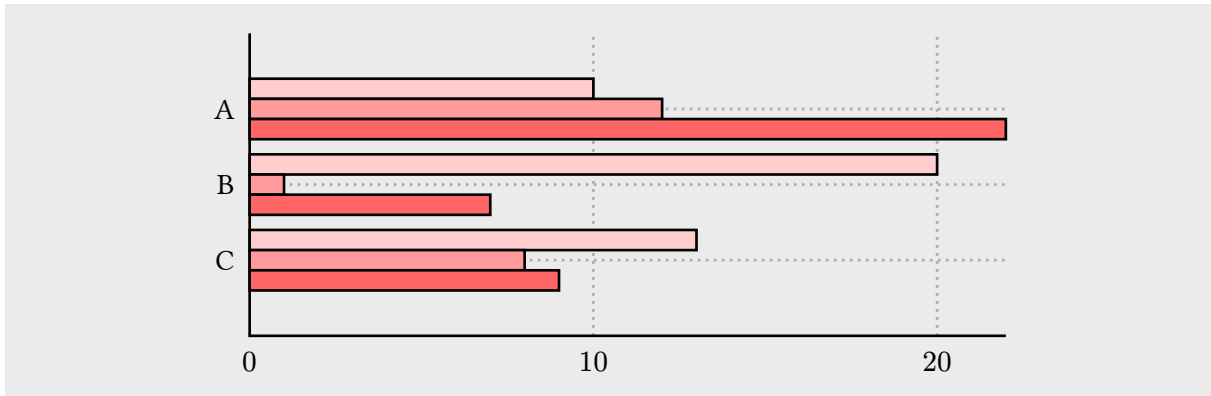
### 6.3.3. Examples – Bar Chart

#### 6.3.3.1. Basic



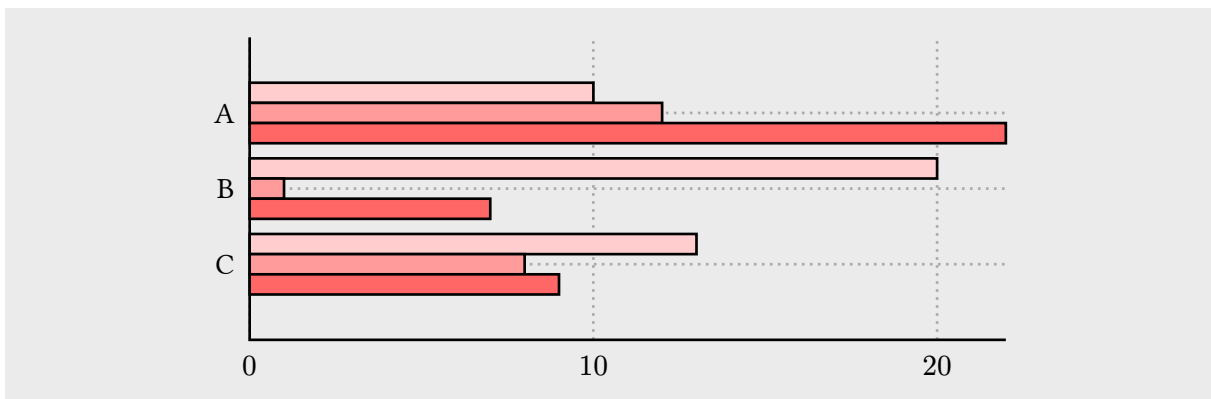
```
import cetz.chart
let data = (("A", 10), ("B", 20), ("C", 13))
chart.barchart(size: (10, auto), x-tick-step: 10, data)
```

### 6.3.3.2. Clustered



```
import cetz.chart
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
chart.barchart(size: (10, auto), mode: "clustered",
               x-tick-step: 10, value-key: (..range(1, 4)), data)
```

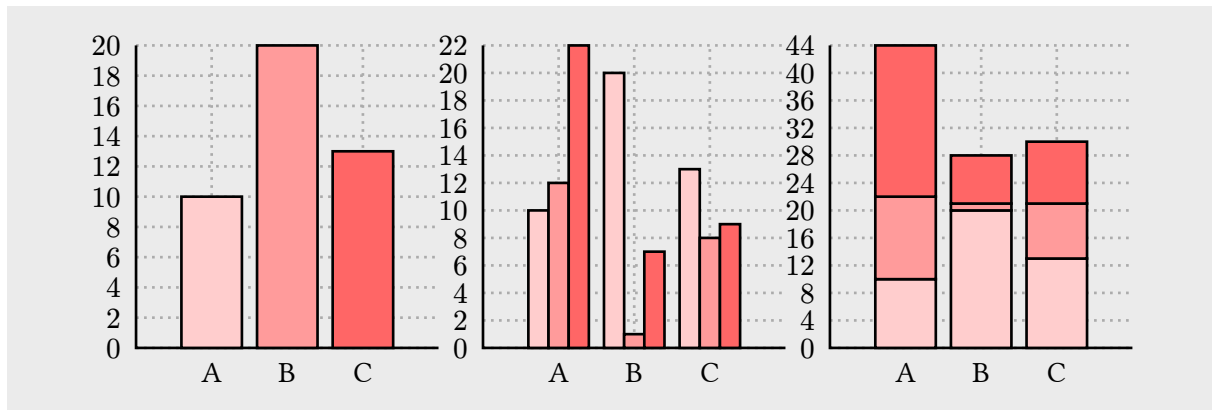
### 6.3.3.3. Stacked



```
import cetz.chart
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
chart.barchart(size: (10, auto), mode: "clustered",
               x-tick-step: 10, value-key: (..range(1, 4)), data)
```

## 6.3.4. Examples – Column Chart

### 6.3.4.1. Basic, Clustered and Stacked



```
import cetz.chart
// Left
let data = (("A", 10), ("B", 20), ("C", 13))
group(name: "a", {
  anchor("default", (0,0))
  chart.columnchart(size: (auto, 4), data)
})
// Center
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
set-origin("a.bottom-right")
group(name: "b", anchor: "bottom-left", {
  anchor("default", (0,0))
  chart.columnchart(size: (auto, 4),
    mode: "clustered", value-key: (1,2,3), data)
})
// Right
let data = (("A", 10, 12, 22), ("B", 20, 1, 7), ("C", 13, 8, 9))
set-origin("b.bottom-right")
group(name: "c", anchor: "bottom-left", {
  anchor("default", (0,0))
  chart.columnchart(size: (auto, 4),
    mode: "stacked", value-key: (1,2,3), data)
})
```

### 6.3.5. Styling

Charts share their axis system with plots and therefore can be styled the same way, see Section 6.2.5.

#### 6.3.5.1. Default barchart Style

```
(axes: (tick: (length: 0)))
```

#### 6.3.5.2. Default columnchart Style

```
(axes: (tick: (length: 0)))
```

## 6.4. Palette

A palette is a function that returns a style for an index. The palette library provides some predefined palettes.

- `new()`

### 6.4.1. new

Define a new palette

A palette is a function in the form `index -> style` that takes an index (int) and returns a canvas style dictionary. If passed the string "len" it must return the length of its styles.

#### 6.4.1.1. Parameters

```
new(
  stroke: stroke,
  fills: array
) -> function
```

**stroke**    stroke

Single stroke style.

**fills**    array

List of fill styles.

#### 6.4.2. List of predefined palettes

- gray



- red



- blue



- rainbow



- tango-light



- tango



- tango-dark



### 6.5. Angle

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

- `angle()`

#### 6.5.1. angle

Draw an angle between origin-a and origin-b Only works for coordinates with  $z = 0$ !

##### anchors:

**start**    Arc starting point

**end**    Arc end point

**origin**    Arc origin

**label**    Label center

### 6.5.1.1. Parameters

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

**origin**    coordinate

Angle corner origin

**a**    coordinate

First coordinate

**b**    coordinate

Second coordinate

**inner**    bool

Draw inner true or outer false angle

Default: true

**label**    content or function or none

Angle label/content or function of the form angle => content that receives the angle and must return a content object

Default: none

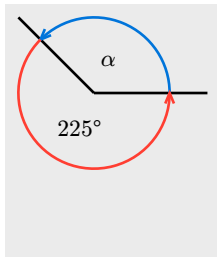
**name**

Default: none

**..style**    style

Angle style





```
import cetz.angle: angle
let (a, b, c) = ((0,0), (-1,1), (1.5,0))
line(a, b)
line(a, c)
set-style(angle: (radius: 1, label-radius: .5), stroke: blue)
angle(a, c, b, label: $alpha$, mark: (end: ">"), stroke: blue)
set-style(stroke: red)
angle(a, b, c, label: n => ${n/1deg} degree$,
      mark: (end: ">"), stroke: red, inner: false)
```

### 6.5.1.2. Default angle Style

```
(
  fill: none,
  stroke: auto,
  radius: 0.5,
  label-radius: 0.25,
  mark: (
    size: 0.15,
    angle: 45deg,
    start: none,
    end: none,
    stroke: auto,
    fill: none,
  ),
)
```

## 6.6. Decorations

Various pre-made shapes and lines.

### 6.6.1. brace

Draw a curly brace between two points.

**Anchors:**

- start** Start coordinate
- end** End coordinate
- spike** Point of the spike
- content** Point to place content at
- center** Center of the enclosing rectangle
- (a-i)** Debug points

#### 6.6.1.1. Parameters

```
brace(
  a: coordinate,
  b: coordinate,
  amplitude: int float,
  pointiness: angle,
  content-offset: int float,
  flip: bool,
  debug: bool,
  name: string none,
  ..style: style
)
```

**a** coordinate

Start point

**b** `coordinate`

End point

**amplitude** `int` or `float`

Height of the brace

Default: `.7`**pointiness** `angle`

How pointy the spike should be. 0deg for maximum pointiness, 90deg for minimum.

Default: `15deg`**content-offset** `int` or `float`

Offset of the content anchor from the spike

Default: `.3`**flip** `bool`

Flip the brace around, same as swapping the start and end points

Default: `false`**debug** `bool`

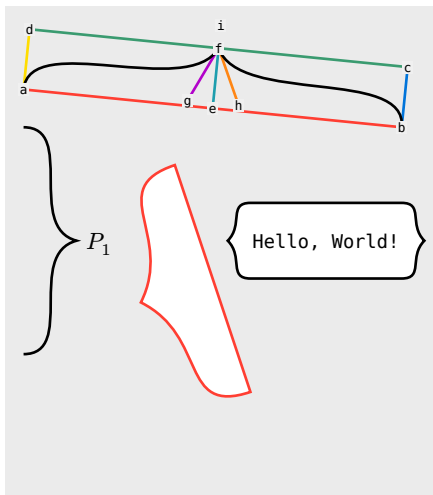
Show debug lines and points

Default: `false`**name** `string` or `none`

Element name

Default: `none`**..style** `style`

Style attributes



```
import cetz.decorations: brace
brace((0, 0), (5, -.5), pointiness: 25deg, amplitude: .8, debug:
true)
brace((0, -.5), (0, -3.5), name: "brace")
content("brace.content", [$P_1$])

// styling can be passed to the underlying `merge-path` call
brace((2, -1), (3, -4), flip: true, amplitude: 1, pointiness: 45deg,
stroke: red, fill: white, close: true)

// as part of another path
set-origin((4, -2))
merge-path({
  brace((+1, .5), (+1, -.5), amplitude: .3, pointiness: 45deg)
  brace((-1, .5), (-1, -.5), amplitude: .3, pointiness: 45deg, flip:
true)
}, fill: white, close: true)
content((0, 0), text(.8em)[Hello, World!])
```

## 7. Advanced Functions

### 7.1. Coordinate

#### 7.1.1. resolve

Resolve a coordinate to a vector

##### 7.1.1.1. Parameters

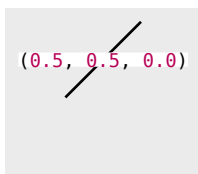
```
resolve(
  ctx: context,
  c: coordinate
) -> vector
```

**ctx**    context

CeTZ context object (see get-ctx or group)

**c**    coordinate

Coordinate



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

### 7.2. Styles

#### 7.2.1. resolve

Resolve the current style root

### 7.2.1.1. Parameters

```
resolve(
  current: style,
  new: style,
  root: none str,
  base: none style
)
```

**current**    style

Current context style (ctx.style).

**new**    style

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

**root**    none or str

Style root element name.

Default: none

**base**    none or style

Base style. For use with custom elements, see `lib/angle.typ` as an example.

Default: none

```
(
  fill: none,
  stroke: 1pt + rgb("#000000"),
  radius: 1,
  mark: (
    size: 0.15,
    angle: 45deg,
    start: none,
    end: none,
    stroke: 1pt + rgb("#000000"),
    fill: none,
  ),
)
```

```
get-ctx(ctx => {
  // Get the current line style
  content((0,0), [#cetz.styles.resolve(ctx.style, (:), root: "line")],
    frame: "rect",
    stroke: none, fill: white)
})
```