

# CeTZ

ein Typst  
Zeichenpaket

Johannes Wolf  
fenjalien

Version 0.2.0

1 Introduction .....	3	4.8 Perpendicular .....	28
2 Usage .....	3	4.9 Interpolation .....	29
2.1 CeTZ Unique Argument Types .....	3	4.10 Function .....	30
2.2 Anchors .....	3	5 Libraries .....	31
2.2.1 Compass Anchors .....	3	5.1 Tree .....	31
3 Draw Function Reference .....	4	5.1.1 tree .....	31
3.1 Canvas .....	4	5.1.2 Node .....	32
3.2 Styling .....	4	5.2 Plot .....	32
3.3 Shapes .....	6	5.2.1 Types .....	32
3.3.1 circle .....	6	5.2.2 plot .....	32
3.3.2 circle-through .....	6	5.2.3 add-anchor .....	36
3.3.3 arc .....	7	5.2.4 add .....	37
3.3.4 mark .....	8	5.2.5 add-hline .....	40
3.3.5 line .....	9	5.2.6 add-vline .....	40
3.3.6 grid .....	10	5.2.7 add-fill-between .....	41
3.3.7 content .....	11	5.2.8 add-contour .....	42
3.3.8 rect .....	12	5.2.9 add-boxwhisker .....	43
3.3.9 bezier .....	12	5.2.10 sample-fn .....	44
3.3.10 bezier-through .....	13	5.2.11 sample-fn2 .....	45
3.3.11 catmull .....	14	5.2.12 Examples .....	45
3.3.12 hobby .....	14	5.2.13 Styling .....	46
3.3.13 merge-path .....	15	5.3 Chart .....	47
3.4 Grouping .....	17	5.3.1 barchart .....	47
3.4.1 intersections .....	17	5.3.2 columnchart .....	48
3.4.2 group .....	17	5.3.3 Examples – Bar Chart .....	50
3.4.3 anchor .....	18	5.3.4 Examples – Column Chart .....	50
3.4.4 copy-anchors .....	19	5.3.5 boxwhisker .....	51
3.4.5 place-anchors .....	19	5.3.6 Styling .....	52
3.4.6 set-ctx .....	19	5.4 Palette .....	52
3.4.7 get-ctx .....	20	5.4.1 new .....	53
3.4.8 for-each-anchor .....	20	5.4.2 List of predefined palettes .....	53
3.4.9 on-layer .....	21	5.5 Angle .....	53
3.4.10 place-marks .....	21	5.5.1 angle .....	53
3.5 Transformations .....	23	5.6 Decorations .....	55
3.5.1 set-transform .....	23	5.6.1 brace .....	55
3.5.2 rotate .....	23	5.6.2 flat-brace .....	56
3.5.3 translate .....	23	6 Advanced Functions .....	58
3.5.4 scale .....	24	6.1 Coordinate .....	58
3.5.5 set-origin .....	24	6.1.1 resolve .....	58
3.5.6 move-to .....	25	6.2 Styles .....	58
3.5.7 set-viewport .....	25	6.2.1 resolve .....	58
4 Coordinate Systems .....	25	6.2.2 Default Style .....	59
4.1 XYZ .....	25	7 Creating Custom Elements .....	60
4.2 Previous .....	26	8 Internals .....	61
4.3 Relative .....	26	8.1 Context .....	61
4.4 Polar .....	26	8.2 Elements .....	61
4.5 Barycentric .....	27		
4.6 Anchor .....	27		
4.7 Tangent .....	28		

## 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 "@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 draw functions override Typst's functions 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.

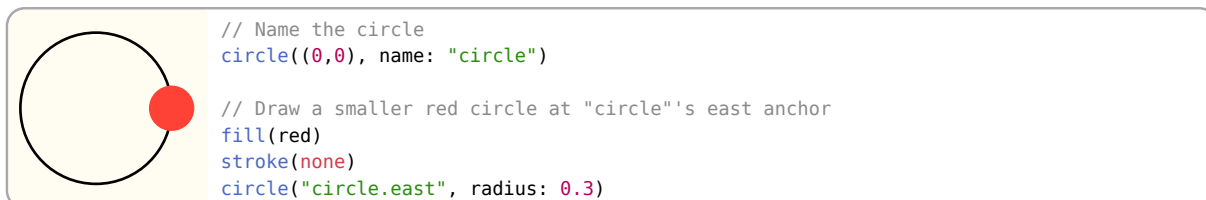
**coordinate** Any coordinate system. See coordinate-systems.

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

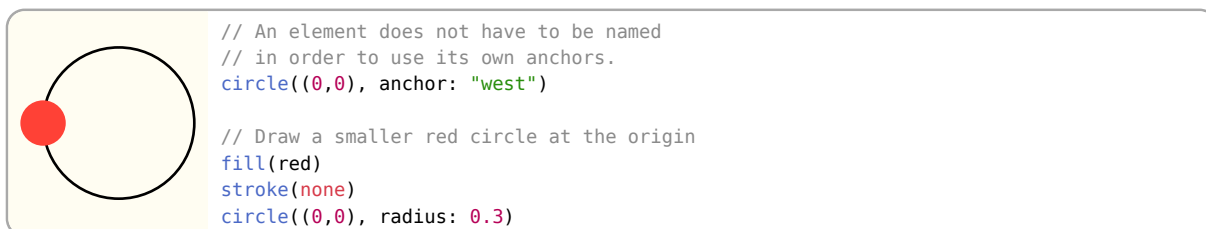
**style** Named arguments (or a dictionary if used for a single argument) of style key-values

### 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 with the `name` argument allow anchors.

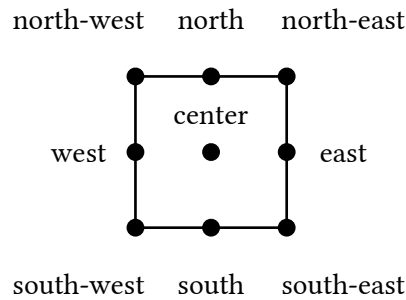


Elements can be placed relative to their own anchors if they have an argument called `anchor`:



#### 2.2.1 Compass Anchors

Some elements support compass anchors. TODO



## 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

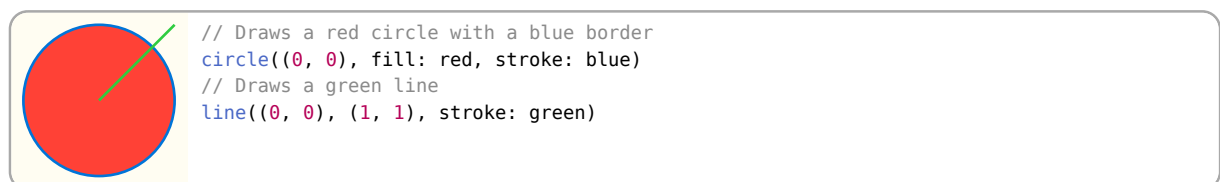
You can style draw elements by passing the relevant named arguments to their draw functions. All elements that draw something have stroke and fill styling unless said otherwise.

**fill** `color` or `none` Default: `none`

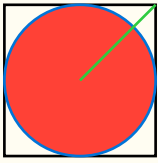
How to fill the drawn element.

**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>



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))

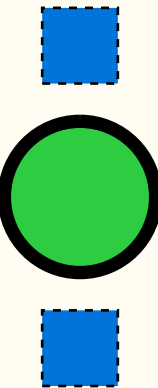
// Draws a green line despite the global stroke is blue
line((0, (1,1), stroke: green)
```

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))
```

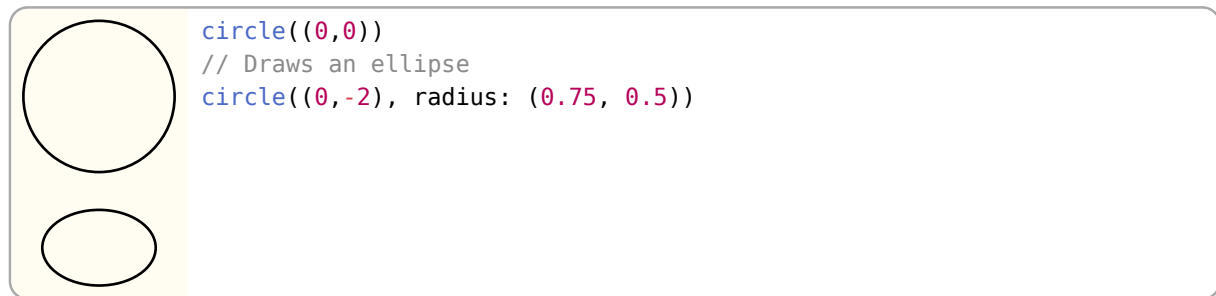


```
// Its a nice drawing okay
set-style(
  rect: (
    fill: red,
    stroke: none
  ),
  line: (
    fill: blue,
    stroke: (dash: "dashed")
  ),
)
rect((0,0), (1,1))
line((0, -1.5), (0.5, -0.5), (1, -1.5), close: true)
circle((0.5, -2.5), radius: 0.5, fill: green)
```

## 3.3 Shapes

### 3.3.1 circle

Draws a circle or ellipse.



#### Parameters

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

**position** coordinate

The position to place the circle on.

**Style Root** circle

#### Style Keys

**radius** number or array

Default: 1

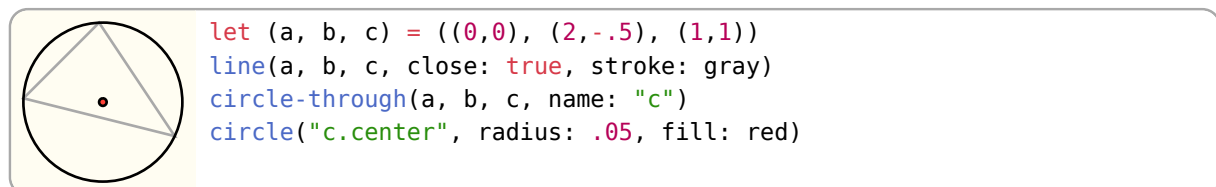
A number that defines the size of the circle's radius. Can also be set to a tuple of two numbers to define the radii of an ellipse, the first number is the x radius and the second is the y radius.

#### Anchors

Supports compass anchors. The "center" anchor is the default.

### 3.3.2 circle-through

Draws a circle through three coordinates



#### 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

### Style Root circle

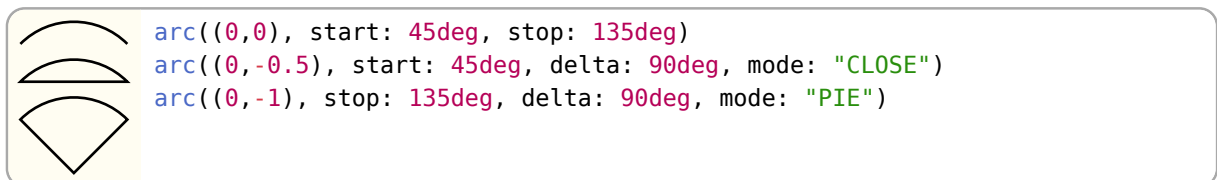
#### Anchors

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.



Note that two of the three angle arguments (start, stop and delta) must be set.

#### 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.

### Style Root arc

#### Style 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.

### Anchors

Supports compass anchors when mode is "PIE"

**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.

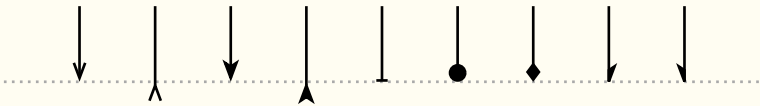
**arc-end** The position of the arc's curve end.

### 3.3.4 mark

Draws a single mark pointing at a target coordinate

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

Or as part of a path based element that supports the mark style key:



```
rotate(-90deg)
set-style(mark: (fill: black))
line((1, -1), (1, 9), stroke: (paint: gray, dash: "dotted"))
line((0, 8), (rel: (1, 0)), mark: (end: "left-harpoon"))
line((0, 7), (rel: (1, 0)), mark: (end: "right-harpoon"))
line((0, 6), (rel: (1, 0)), mark: (end: "<"))
line((0, 5), (rel: (1, 0)), mark: (end: "o"))
line((0, 4), (rel: (1, 0)), mark: (end: "|"))
line((0, 3), (rel: (1, 0)), mark: (end: ">"))
line((0, 2), (rel: (1, 0)), mark: (end: ">"))
set-style(mark: (fill: none))
line((0, 1), (rel: (1, 0)), mark: (end: "<"))
line((0, 0), (rel: (1, 0)), mark: (end: ">"))
```

### 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.

**Style Root** `mark`

**Style Keys**

**symbol** `string`

Default: `">"`

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

**start** `string` or `none` or `array`

Default: `none`

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

**end** `string` or `none` or `array`

Default: `none`

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

**length** `number`

Default: `0.2`

The length of the mark along its direction it is pointing.

**width** `number`

Default: `0.15`

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

**inset** `number`

Default: `0.05`

The distance by which something inside the arrow tip is set inwards.

**scale** `float`

Default: `1`

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

**sep** `number`

Default: `1`

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`

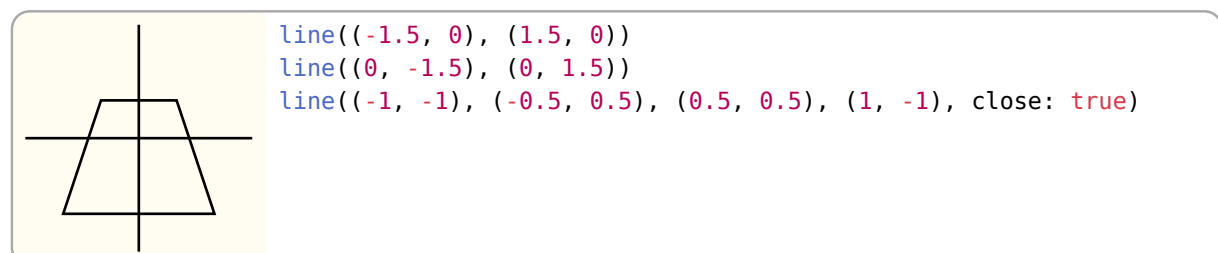
Default: `30`

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.

**Note:** The size of the mark depends on its style values, not the distance between `from` and `to`, which only determine its orientation.

### 3.3.5 line

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



**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

**Style Root** line

**Style Keys**

Supports marks

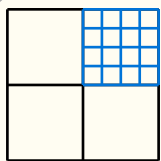
**Anchors**

**start** The line's start position

**end** The line's end position

**3.3.6 grid**

Draw 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)
```

**Style Root** grid

**Anchors**

Supports compass anchors.

**Parameters**

```
grid(
  from: coordinate,
  to: coordinate,
  step: number,
  name: none string,
  help-lines,
  ..style: style
)
```

**from** coordinate

The top left of the grid

**to** coordinate

The bottom right of the grid

**step** number

Default: "1"

Grid spacing.

**help-lines**

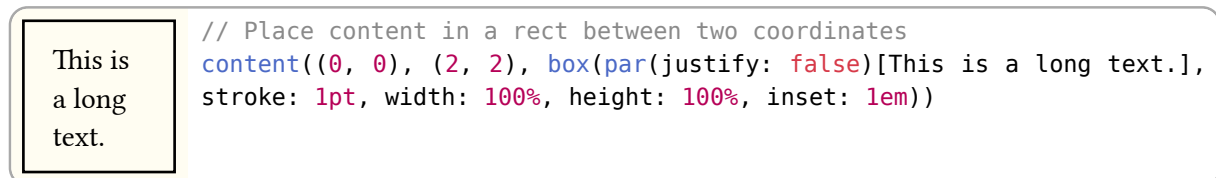
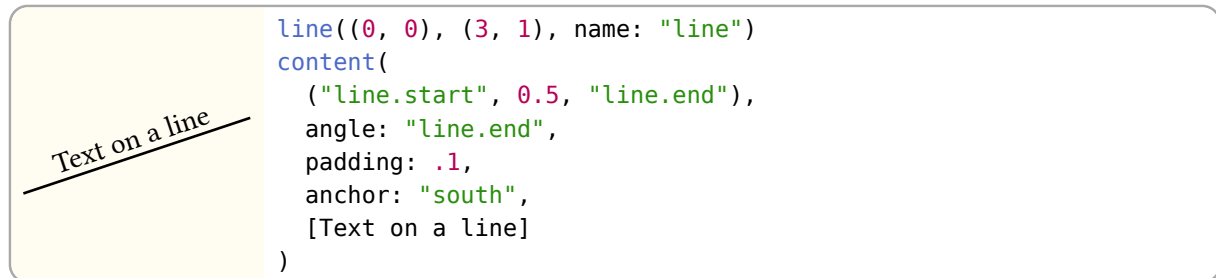
Default: "false"

### 3.3.7 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:



#### 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.

#### Style Root content

##### Style 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`

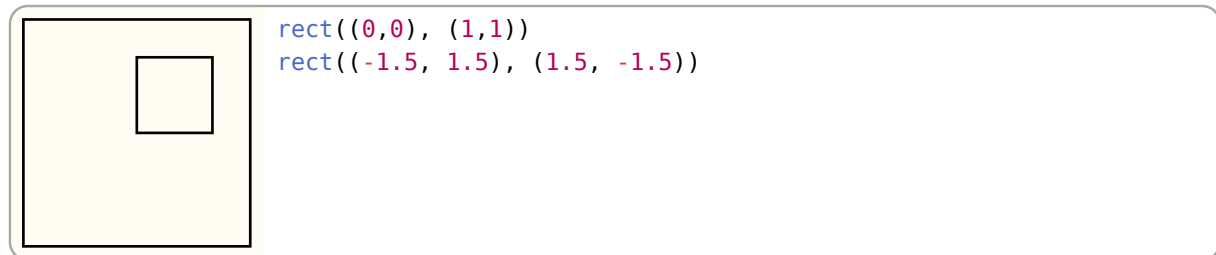
Sets the frame style. Can be none, "rect" or "circle" and inherits the stroke and fill style.

## Anchors

Supports compass anchors.

### 3.3.8 rect

Draws a rectangle between two coordinates.



## Style Root rect

## Anchors

Supports compass anchors.

## Parameters

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

### a coordinate

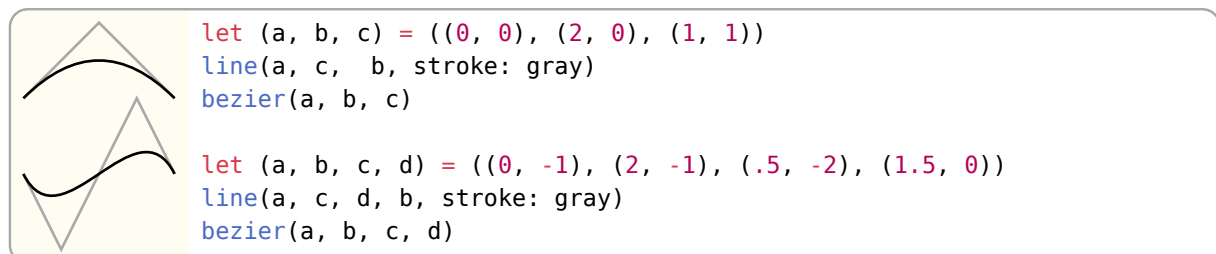
Coordinate of the top left corner of the rectangle.

### b coordinate

Coordinate of the bottom 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)).

### 3.3.9 bezier

Draws a quadratic or cubic bezier curve



## 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.

**Style Root** bezier

**Style Keys**

Supports marks.

**Anchors**

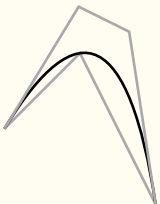
**ctrl-n** nth control point where n is an integer starting at 0

**start** The start position of the curve.

**end** The end position of the curve.

### 3.3.10 bezier-through

Draw 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

Start position

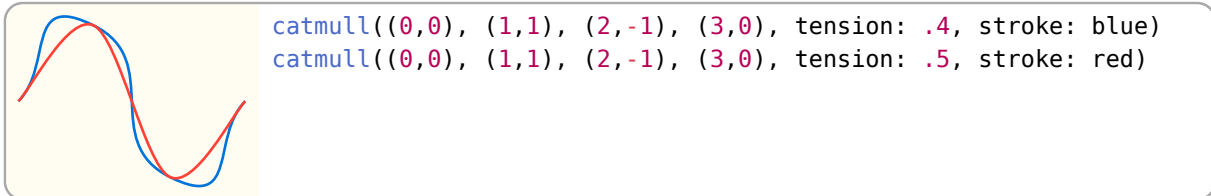
**pass-through** coordinate

Curve mid-point

**end** `coordinate`  
End coordinate

### 3.3.11 catmull

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



#### 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.

**Style Root** `catmull`

#### Style Keys

**tension** `float`

Default: `0.5`

I need a description

Supports marks.

#### Anchors

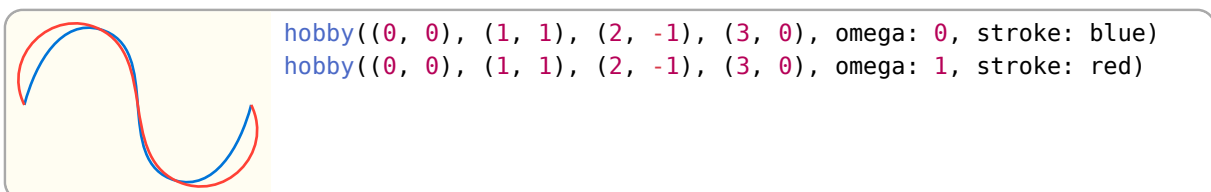
**start** The position of the start of the curve.

**end** The position of the end of the curve.

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

### 3.3.12 hobby

Draws a Hobby curve through a set of points.



## 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 straight line between the start and end of the curve.

## Style Root hobby

### Style Keys

Supports marks.

**omega** idk Default: none

The curve's curlyness

**rho** idk Default: none

## Anchors

**start** The position of the start of the curve.

**end** The position of the end of the curve.

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

### 3.3.13 merge-path

Merges two or more paths by concatenating 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))
})
```

## Parameters

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

**body** elements

Elements with paths to be merged together.

**close** bool

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

Default: "false"

## Aliases

**start** The start of the merged path.

**end** The end of the merged path.





## 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.

## Style Root group

## Style 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).

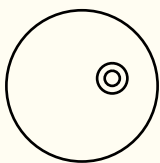
**anchors** Supports compass anchors. These are created based on the axis aligned bounding box of all the child elements of the group.

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".

### 3.4.3 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

### 3.4.4 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.

### 3.4.5 place-anchors

TODO: Not writing the docs for this as it should be removed in place of better anchors before 0.2 Place multiple anchors along a path

#### Parameters

```
place-anchors(
  path: drawable,
  ..anchors: array,
  name
)
```

**path** `drawable`

Single drawable

**..anchors** `array`

List of anchor dictionaries of the form `(pos: <float>, name: <string>)`, where `pos` is a relative position on the path from 0 to 1.

- `name: (auto,string)`: If `auto`, take the name of the passed drawable. Otherwise sets the elements name

**name**

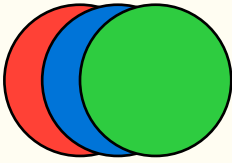
Default: `"auto"`

### 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

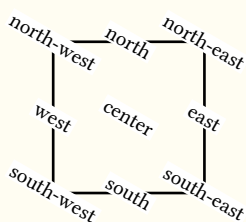
`get-ctx`(callback: function)

**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`

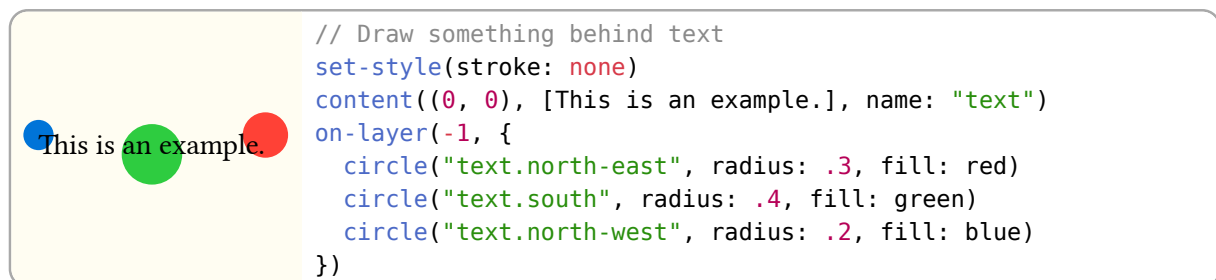
A function that takes the anchor name and can return elements.

### 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.



## 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.

### 3.4.10 place-marks

TODO: Not writing the docs for this as it should be removed in place of better anchors before 0.2 Place one or more marks along a path

Mark items must get passed as positional arguments. A mark-item is an dictionary of the format: (mark: "<symbol>", pos: <float>), where the position pos is a relative position from 0 to 1 along the path.

## Parameters

```
place-marks(  
  path: drawable,  
  ..marks-style: mark-item style,  
  name: none string  
)
```

**path** drawable

A single drawable

**..marks-style** mark-item or style

Positional mark-items and style key-value pairs

**name** none or string

Element name

Default: "none"

### 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.



##### Parameters

`rotate(..angles: angle)`

**..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 to TODO

#### 3.5.3 translate

Translates the transformation matrix by the given vector or dictionary.



## Parameters

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

**vec** vector or dictionary

The vector to translate by. A dictionary can be given instead with optional keys x, y and z to translate in the relevant axis.

**pre** bool

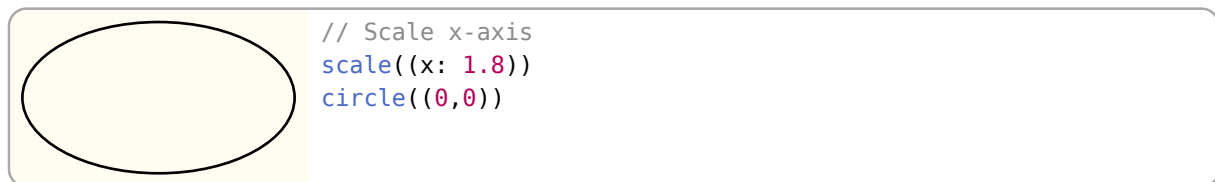
Default: "true"

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).



## Parameters

```
scale(factor: float dictionary)
```

**factor** float or dictionary

A float to scale the transformation matrix by. A dictionary with optional keys x, y and z can also be given to scale in the respective directions.

### 3.5.5 set-origin

Sets the given position as the origin



## 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.

#### 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

#### 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` 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)`.




```

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

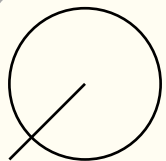
// Implicit form
line((0, -2), (1, -2))
line((0, -2), (0, -1, 0))
line((0, -2), (0, -2, 1))

```



## 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** `bool`

(default: `true`)

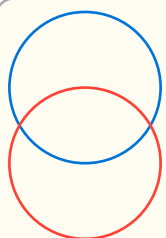
When false the previous position will not be updated.

**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.



```

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

```

## 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.

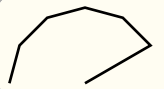
**radius** `number` or `<length>` or `<array of length or number>`

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.



```
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))
```

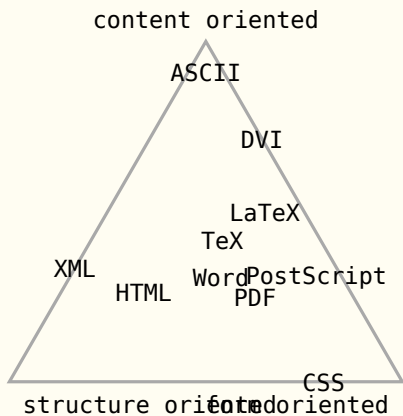
## 4.5 Barycentric

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors  $v_1, v_2, \dots, v_n$  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  $a$ .



```

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-west"),
  ("form", "north-east")
) {
  content(c, box(c + " oriented", inset: 5pt), 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)
}

```

## 4.6 Anchor

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

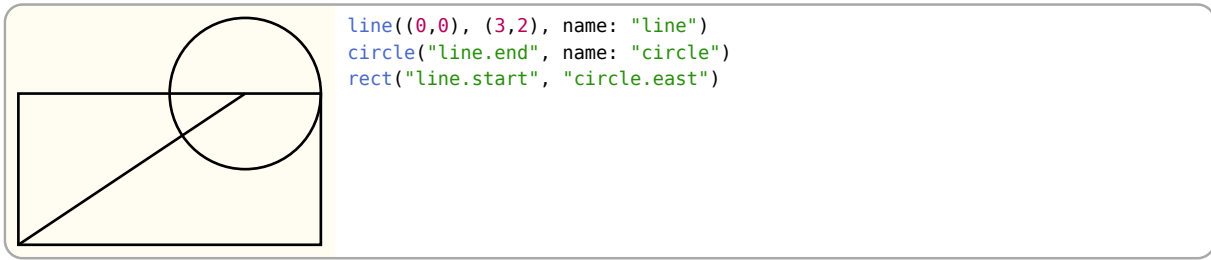
**name** `string`

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".



## 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.

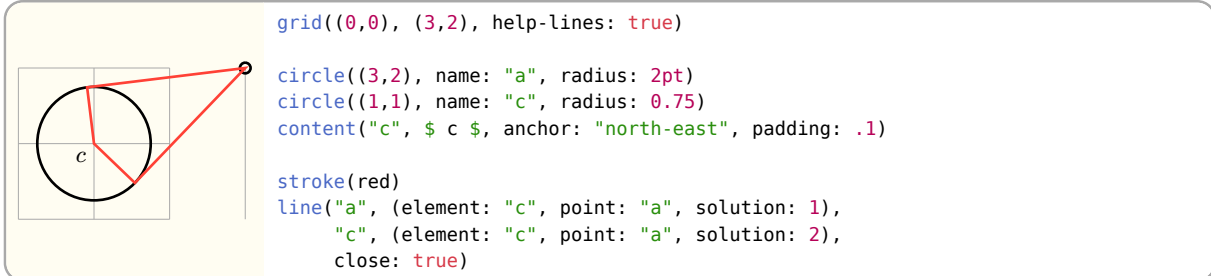
**point** coordinate

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.



## 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  $q$ .

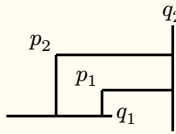
**horizontal** coordinate

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 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

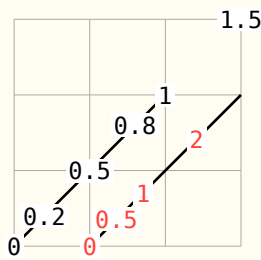
(default: 0deg)

**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).



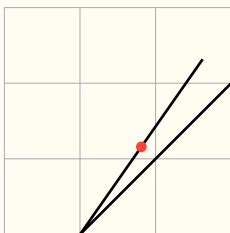
```

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

line((0,0), (2,2))
for i in (0, 0.2, 0.5, 0.8, 1, 1.5) { /* Relative distance */
  content(((0,0), i, (2,2)),
    box(fill: white, inset: 1pt, [#i]))
}

line((1,0), (3,2))
for i in (0, 0.5, 1, 2) { /* Absolute distance */
  content((a: (1,0), number: i, abs: true, b: (3,2)),
    box(fill: white, inset: 1pt, text(red, [#i])))
}

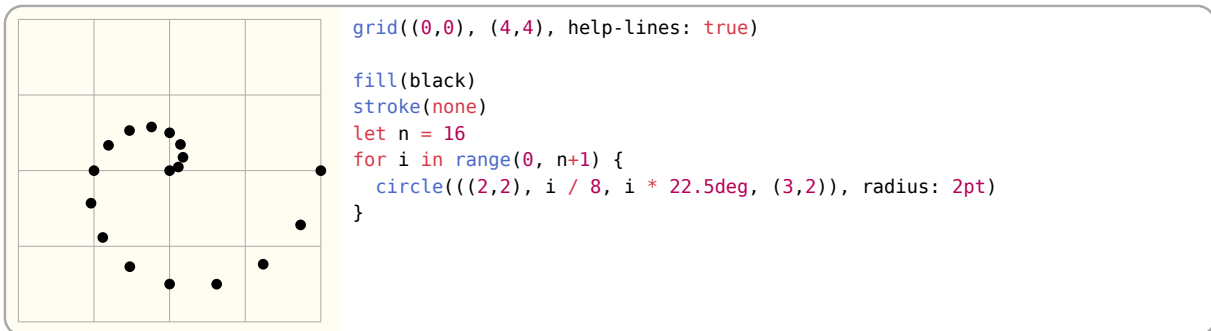
```



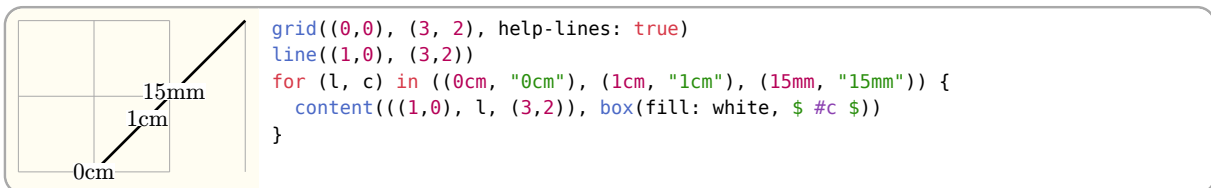
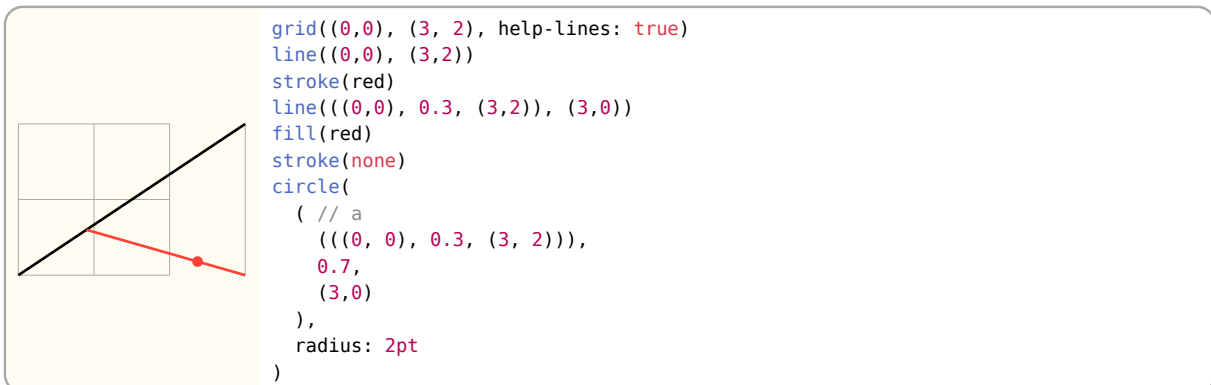
```

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), 0.5, 10deg, (3, 2)), radius: 2pt)

```



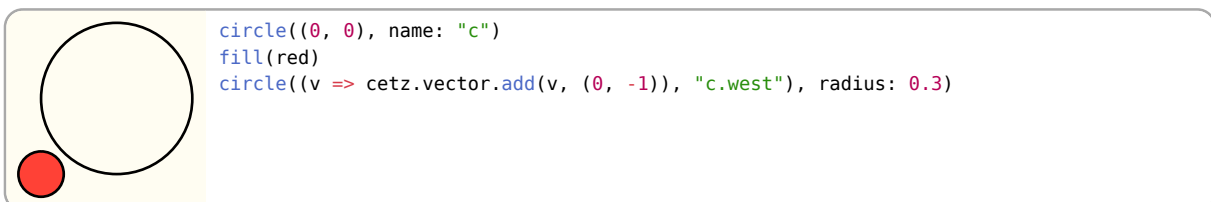
You can even chain them together!



## 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.



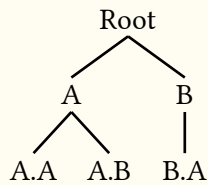
## 5 Libraries

### 5.1 Tree

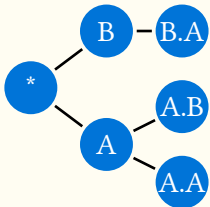
The tree library allows the drawing diagrams with simple tree layout algorithms

#### 5.1.1 tree

Lays out and renders tree nodes.



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



```
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))
  }
)
```

### Parameters

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

**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 contents will be drawn.

<b>draw-edge</b>	<code>auto</code> or <code>function</code>	Default: <code>"auto"</code>
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 <code>((source-name, target-name, target-node) =&gt; elements)</code> . If <code>auto</code> is given, a straight line will be drawn between nodes.		
<b>direction</b>	<code>string</code>	Default: <code>"\down"</code>
A string describing the direction the tree should grow in ("up", "down", "left", "right")		
<b>parent-position</b>	<code>string</code>	Default: <code>"center"</code>
Positioning of parent nodes (begin, center, end)		
<b>grow</b>	<code>float</code>	Default: <code>1</code>
Depth grow factor (default 1)		
<b>spread</b>	<code>float</code>	Default: <code>1</code>
Sibling spread factor (default 1)		
<b>name</b>		Default: <code>none</code>
<b>..style</b>		

### 5.1.2 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.

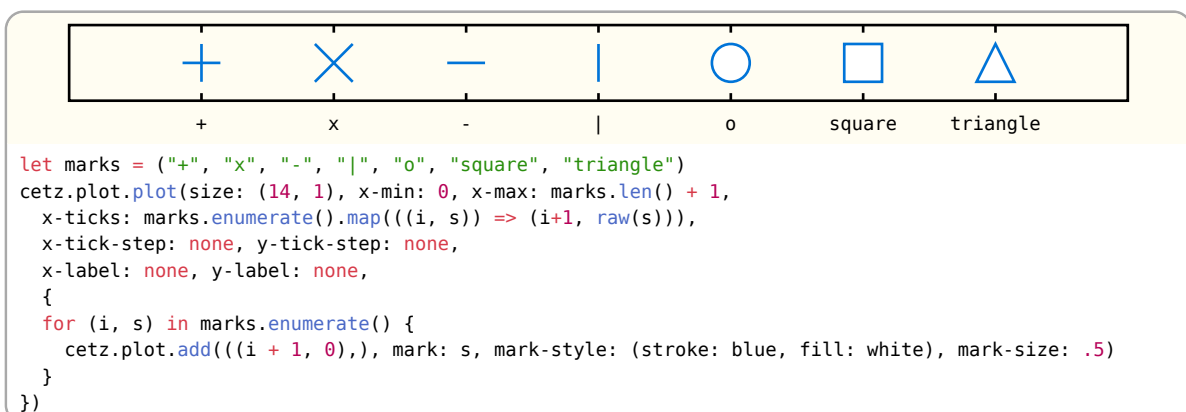
## 5.2 Plot

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  $(-\text{calc.pi}, \text{calc.pi})$  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:



### 5.2.2 plot

Create a plot environment

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





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

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 "x2" and "y2", if set (if unset, "x2" mirrors "x" and "y2" mirrors "y"). You can use any axis name and as many axes as you want for plotting data, but only the predefined axes (x, y, x2, y2) are displayed with ticks.

To draw elements inside a plot, using the plot's coordinate system, use the `plot.add-annotation(...)` function.

## Options

The following options are supported per axis and must be prefixed by an axis name: `<axis-name>-<option>`, e.g. `x-min: 0` or `y-label: [y]`.

**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 > than max, the axis' direction is swapped

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

Axis upper domain value. If this is set < than min, the axis' direction is swapped

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

Set the axis aspect ratio to be fixed to the aspect ratio of the given axis. This can be useful to force one axis to grow or shrink with another one. You can only "lock" two axes of different orientation (horizontal).

**horizontal** `bool` Default: "auto"

If true, values on this axis are drawn horizontally, otherwise values get drawn vertically.

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

Increment between tick marks on the axis, starting at 0. If set to auto, a matching increment is calculated. When set to none, tick marks are disabled.

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

Like tick-step, but for minor tick marks.

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

List of custom tick marks in addition to those generated by tick-step and minor-tick-step. This argument supports an array of float on where to place marks, or an array of (`<float>`, `<content>`) tuples, for setting custom tick mark labels per mark. Examples: (1, 2, 3) or ((1, [One]), (2, [Two]), (3, [Three]))

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

Specifies the tick label formatting or a custom formatter function. The following predefined formats are supported:

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

**"sci"** Scientific formatting with  $\times 10^n$  used as exponent syntax

**number => content** A function that takes a number and returns content that gets used as tick label

- 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.

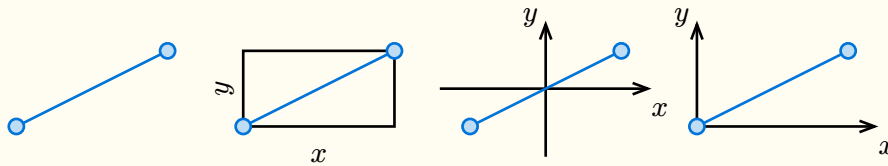
## Parameters

```
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.add-annotation`, 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\""`

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

**"scientific"** Frame plot area using a rect 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.**"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")

cetz.plot.plot(axis-style: none, ..opts, data)
set-origin((3,0))
cetz.plot.plot(axis-style: "scientific", ..opts, data)
set-origin((3,0))
cetz.plot.plot(axis-style: "school-book", ..opts, data)
set-origin((3,0))
cetz.plot.plot(axis-style: "left", ..opts, data)
```

**name** `string`Default: `"none"`

The plots element name to be used when referring to anchors

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

Style used for drawing plot graphs This style gets inherited by all plots and supports palette functions.

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

Style used for drawing plot marks. This style gets inherited by all plots and supports palette functions.

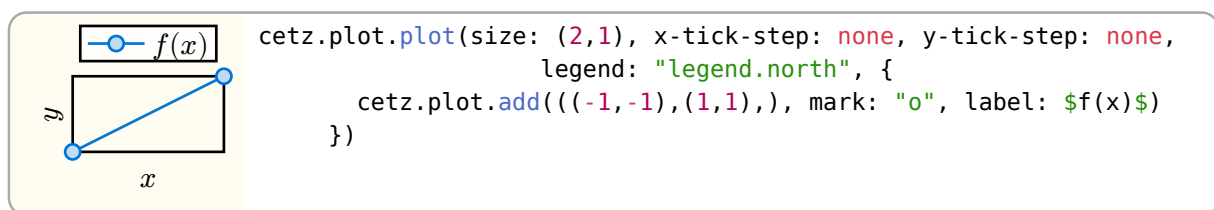
**fill-below** `bool`Default: `"true"`

If true, fill functions below the axes (draw axes above filled plots), if false filled areas get drawn above the plots axes.

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

Position to place the legend at. The legend is drawn if at least one plot with `label: ..` set to a value `!= none` exists. The following anchors are considered optimal for legend placement:

- `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`



If set to `auto`, the placement of the legend style (**Style Root** legend) gets used. If set to a coordinate, that coordinate, relative to the plots origin is used for placing the legend group.

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

Anchor of the legend group to use as its origin. If set to `auto` and `legend` 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.

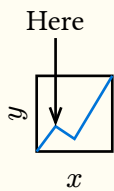
**..options** `any`

Axis options, see *options* above.

### 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(x-tick-step: none, y-tick-step: none, name: "plot", {
  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"), note that both axes must exist, as add-anchors does not create axes on demand.

### 5.2.4 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.

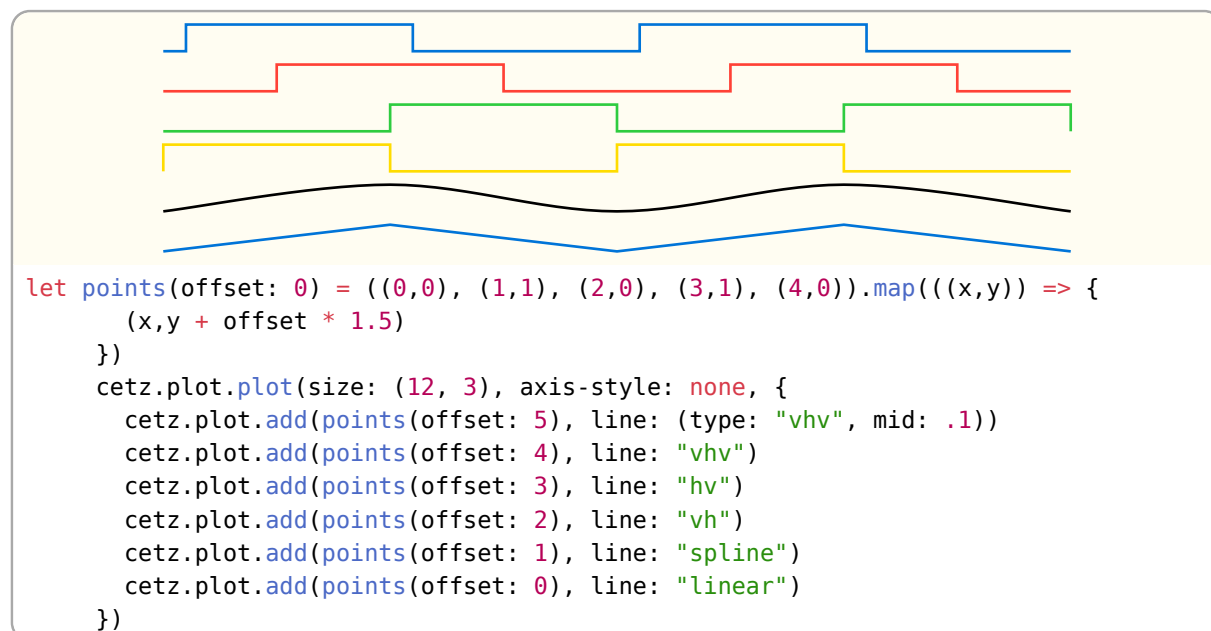
<b>hypograph</b>	bool	Default: "false"
Fill hypograph; uses the hypograph style key for drawing		
<b>epigraph</b>	bool	Default: "false"
Fill epigraph; uses the epigraph style key for drawing		
<b>fill</b>	bool	Default: "false"
Fill the shape of the plot		
<b>fill-type</b>	string	Default: "\"axis\""
Fill type:		
"axis" Fill the shape to $y = 0$		
"shape" Fill the complete shape		
<b>style</b>	style	Default: "( : )"
Style to use, can be used with a palette function		
<b>mark</b>	string	Default: "none"
Mark symbol to place at each distinct value of the graph. Uses the mark style key of style for drawing.		
<b>mark-size</b>	float	Default: ".2"
Mark size in canvas units		
<b>mark-style</b>		Default: "( : )"
<b>samples</b>	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.		
<b>sample-at</b>	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**"vhv"** 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.

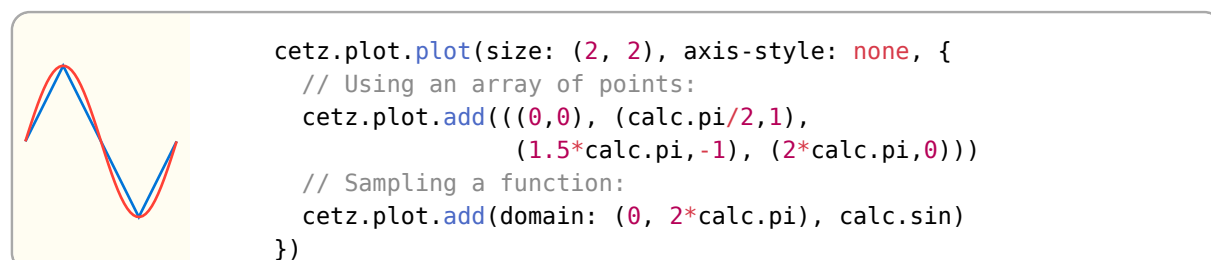
If the value is a dictionary, the type must be supplied via the type key. The following extra attributes are supported:

**"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.**axes** `axes`Default: `"(\\"x\\", \\"y\\")"`

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

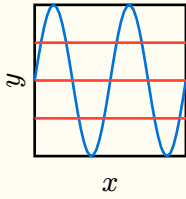
**label** `none` or `content`Default: `"none"`

Legend label to show for this plot.

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

### 5.2.5 add-hline

Add horizontal lines at one or more y-values. The lines start and end point is at its 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: number,
  axes: array,
  style: style,
  label: none content
)
```

**..y** number

Y axis value(s) to add a line at

**axes** array

Name of the axes to use for plotting

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

**style** style

Style to use, can be used with a palette function

Default: `"(:)"`

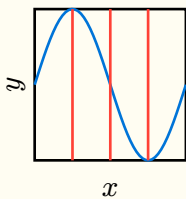
**label** none or content

Legend label to show for this plot.

Default: `"none"`

### 5.2.6 add-vline

Add vertical lines at one or more x-values. The lines start and end point is at its 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: number,
  axes: array,
  style: style,
  label: none content
)
```

**..x** number

X axis values to add a line at

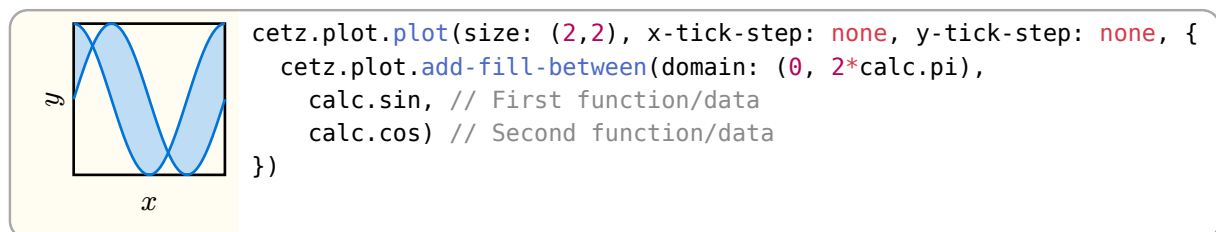


<b>axes</b>	array	Default: <code>"(\\"x\\", \\"y\\")"</code>
	Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!	
<b>style</b>	style	Default: <code>"(:)"</code>
	Style to use, can be used with a palette function	
<b>label</b>	none or content	Default: <code>"none"</code>
	Legend label to show for this plot.	

### 5.2.7 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.



### 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

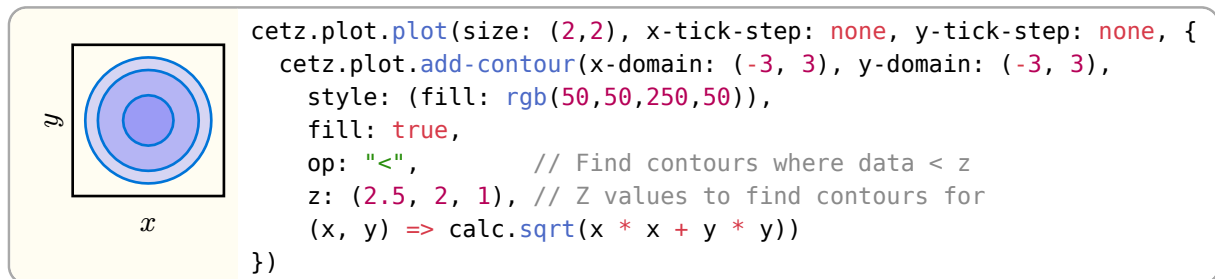
Default: `"()"`

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

<b>line</b>	string or dictionary	Default: <code>"linear"</code>
	Line type to use, see <code>add()</code> .	
<b>axes</b>	array	Default: <code>"(x", "y)"</code>
	Name of the axes to use for plotting.	
<b>label</b>	none or content	Default: <code>"none"</code>
	Legend label to show for this plot.	
<b>style</b>	style	Default: <code>"(:)"</code>
	Style to use, can be used with a palette function.	

### 5.2.8 add-contour

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



#### 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  $(x, y) \Rightarrow z$  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  $z$  ( $z \geq 0$ ) or below  $z$  ( $z < 0$ ) get plotted. If you specify multiple  $z$  values, they get plotted in the order of specification.

**x-domain** domain

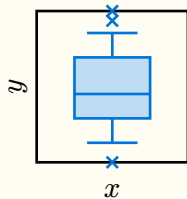
Default: `"(0, 1)"`

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

<b>y-domain</b>	domain	Default: "(0, 1)"
Y axis domain used if data is a function, see x-domain.		
<b>x-samples</b>	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.		
<b>y-samples</b>	int	Default: "25"
Y axis domain samples ( $2 < n$ )		
<b>interpolate</b>	bool	Default: "true"
Use linear interpolation between sample values which can improve the resulting plot, especially if the contours are curved.		
<b>op</b>	auto or string or function	Default: "auto"
Z value comparison operator:		
">", ">=", "<", "<=", "!=", "==" Use the operator for comparison of z to the values from data.		
<b>auto</b> Use ">=" for positive z values, "<=" for negative z values.		
<b>function</b> 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.		
<b>axes</b>	axes	Default: "(\\"x\\", \\"y\\")"
Name of the axes to use for plotting.		
<b>style</b>	style	Default: "(:)"
Style to use for plotting, can be used with a palette function. Note that all z-levels use the same style!		
<b>fill</b>	bool	Default: "false"
Fill each contour		
<b>limit</b>	int	Default: "50"
Limit of contours to create per z value before the function panics		

### 5.2.9 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 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

### 5.2.10 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  $(x) \Rightarrow y$  or  $(t) \Rightarrow (x, y)$ , 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 outside the specified domain are legal.

**5.2.11 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  $(x, y) \Rightarrow z$  with all values being numbers.

**x-domain** domain

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

**y-domain** domain

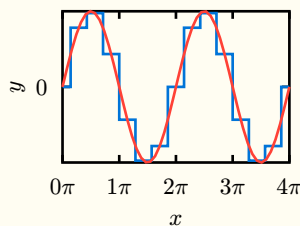
Domain used as bounding interval for sampling point's  $y$  values.

**x-samples** int

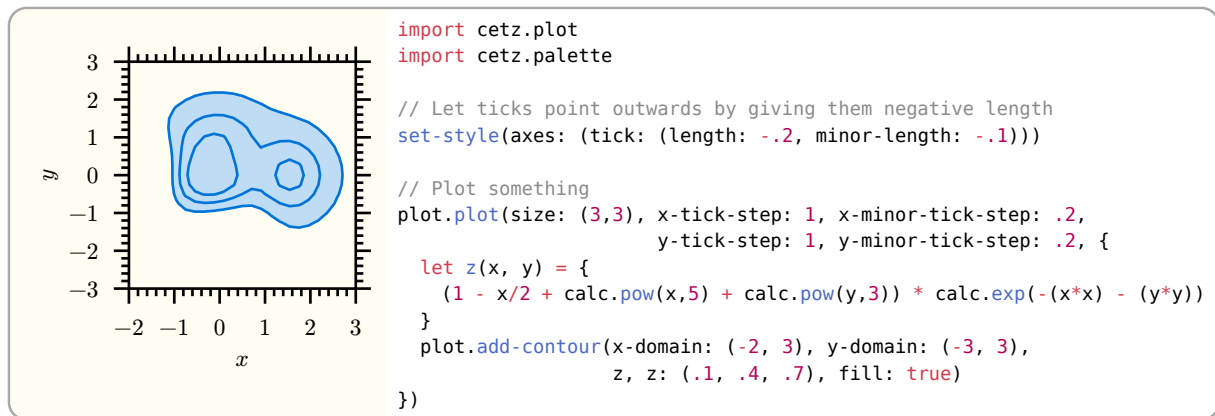
Number of samples in the  $x$ -domain.

**y-samples** int

Number of samples in the  $y$ -domain.

**5.2.12 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: "vhv", style: (mark: (stroke: blue)))
  plot.add(domain: (0, 4*calc.pi), calc.sin)
})
```



### 5.2.13 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

```
(
  fill: none,
  stroke: luma(0%),
  label: (offset: 0.2, anchor: auto),
  tick: (
    fill: none,
    stroke: luma(0%),
    length: 0.1,
    minor-length: 0.08,
    label: (offset: 0.2, angle: 0deg, anchor: auto),
  ),
  grid: (
    stroke: (paint: luma(66.67%), dash: "dotted"),
    fill: none,
  ),
)
```

#### Default school-book Style

```
(
  fill: none,
  stroke: luma(0%),
  label: (offset: 0.2, anchor: auto),
  tick: (
    fill: none,
```

```

    stroke: luma(0%),
    length: 0.1,
    minor-length: 0.08,
    label: (offset: 0.1, angle: 0deg, anchor: auto),
  ),
  grid: (
    stroke: (paint: luma(66.67%), dash: "dotted"),
    fill: none,
  ),
  mark: (end: ">"),
  padding: 0.4,
)

```

## 5.3 Chart

With the chart library it is easy to draw charts.

Supported charts are:

- `barchart(...)` and `columnchart(...)`: A chart with horizontal/vertical 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
- `boxwhisker(...)`: A box-plot chart

### 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 5.3.3.

**Style root:** `barchart`.

#### 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-decimals: int,
  x-format: string function,
  x-min: number auto,
  x-max: number 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),)
```

<b>label-key</b>	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.		
<b>value-key</b>	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.		
<b>mode</b>	string	Default: "\"basic\""
Chart mode:		
<ul style="list-style-type: none"> <li>• "basic" – Single bar per data row</li> <li>• "clustered" – Group of bars per data row</li> <li>• "stacked" – Stacked bars per data row</li> <li>• "stacked100" – Stacked bars per data row relative to the sum of the row</li> </ul>		
<b>size</b>	array	Default: "(1, auto)"
Chart size as width and height tuple in canvas unist; height can be set to auto.		
<b>bar-width</b>	float	Default: ".8"
Size of a bar in relation to the charts height.		
<b>bar-style</b>	style or function	Default: "palette.red"
Style or function (idx => style) to use for each bar, accepts a palette function.		
<b>x-tick-step</b>	float	Default: "auto"
Step size of x axis ticks		
<b>x-ticks</b>	array	Default: "()"
List of tick values or value/label tuples		

### Example

(1, 5, 10) or ((1, [One]), (2, [Two]), (10, [Ten]))

<b>x-unit</b>	content or auto	Default: "auto"
Tick suffix added to each tick label		
<b>x-decimals</b>	int	Default: "1"
Number of x axis tick decimals		
<b>x-format</b>	string or function	Default: "\"float\""
X axis tick format, "float", "sci" or a callback of the form float => content.		
<b>x-min</b>	number or auto	Default: "auto"
X axis minimum value		
<b>x-max</b>	number or auto	Default: "auto"
X axis maximum value		
<b>x-label</b>	content or none	Default: "none"
X axis label		
<b>y-label</b>	content or none	Default: "none"
Y axis label		

### 5.3.2 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 5.3.4.

**Style root:** columnchart.



## 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-format: string function,
  y-decimals: int,
  y-label: content none,
  y-min: number auto,
  y-max: number auto
)
```

**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-width** float

Default: ".8"

Size of a bar in relation to the charts height.

**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-tick-step** float

Default: "auto"

Step size of y axis ticks

**y-ticks** array

Default: "()"

List of tick values or value/label tuples

**Example**

(1, 5, 10) or ((1, [One]), (2, [Two]), (10, [Ten]))

**y-unit** content or auto

Default: "auto"

Tick suffix added to each tick label

**y-format** string or function

Default: "\\float\\"

Y axis tick format, "float", "sci" or a callback of the form float =&gt; content.

**y-decimals** int

Default: "1"

Number of y axis tick decimals

**y-label** content or none

Default: "none"

Y axis label

**y-min** number or auto

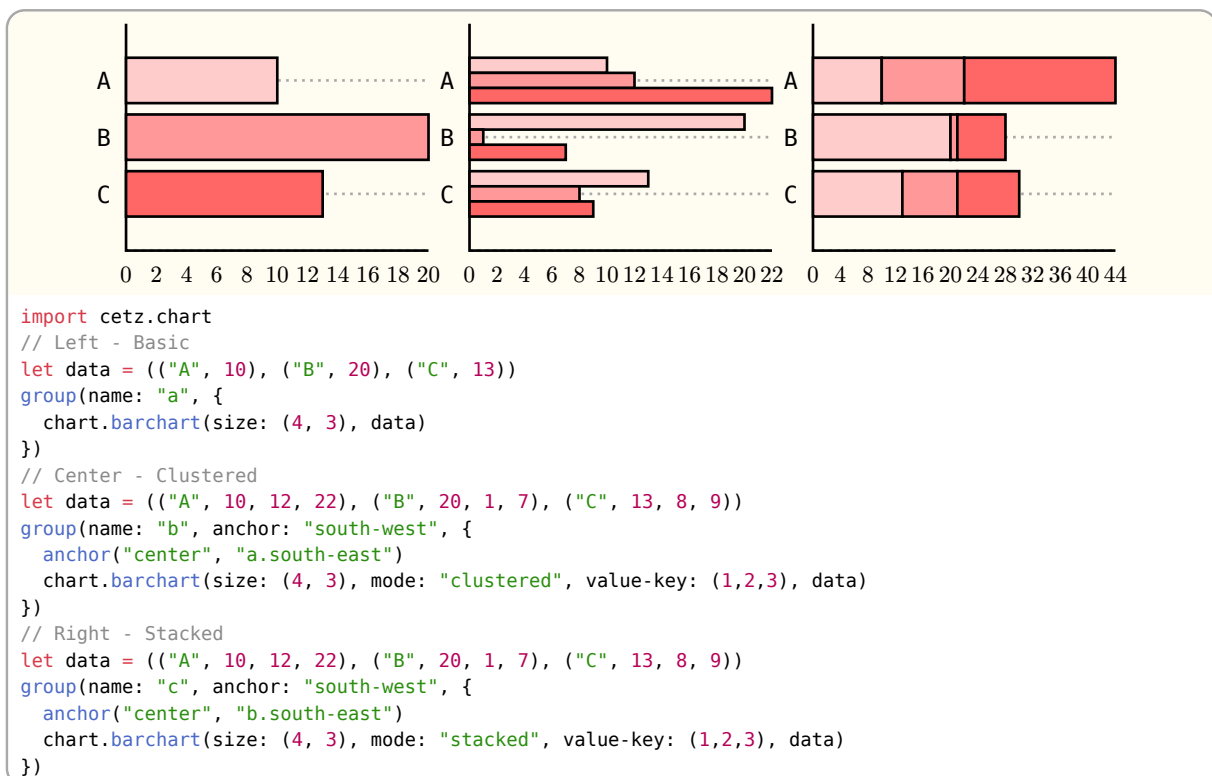
Default: "auto"

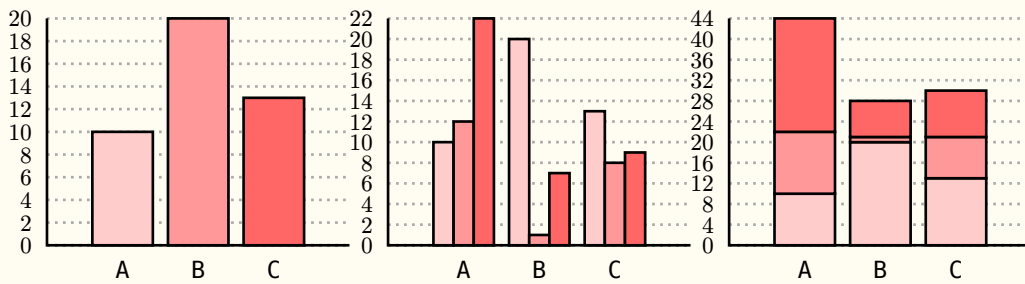
Y axis minimum value

**y-max** number or auto

Default: "auto"

Y axis maximum value

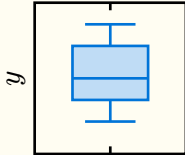
**5.3.3 Examples – Bar Chart****5.3.4 Examples – Column Chart****Basic, Clustered and Stacked**



```
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)
})
```

### 5.3.5 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: none,
  (x: 1, min: 15, max: 60,
    q1: 25, q2: 35, q3: 50))
```

### Parameters

```
boxwhisker(
  data: array dictionary,
  size,
  y-min,
  y-max,
  label-key: integer string,
  box-width: float,
  whisker-width: float,
  mark: string,
  mark-size: float,
  ..arguments: 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
- `y-min (float)` : Lower end of y-axis range. If `auto`, defaults to lowest outlier or lowest min.
- `y-max (float)` : Upper end of y-axis range. If `auto`, defaults to greatest outlier or greatest max.

**size** `Default: "(1, auto)"`

**y-min** `Default: "auto"`

**y-max** `Default: "auto"`

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

Index in the array where labels of each entry is stored

**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

**..arguments** `any`

Additional arguments are passed to `plot.plot`

### 5.3.6 Styling

Charts share their axis system with plots and therefore can be styled the same way, see Section 5.2.13.

#### Default `barchart` Style

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

#### Default `columnchart` Style

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

#### Default `boxwhisker` Style

```
(axes: (tick: (length: -0.1)), grid: none)
```

## 5.4 Palette

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

- `new()`

### 5.4.1 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.

#### Parameters

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

**stroke**    stroke

Single stroke style.

**fills**    array

List of fill styles.

### 5.4.2 List of predefined palettes

- gray



- red



- blue



- rainbow



- tango-light



- tango



- tango-dark

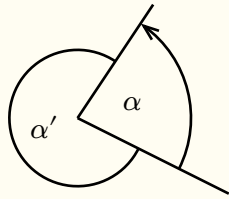


## 5.5 Angle

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.

**Aliases:**

**"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.

**Default angle Style**

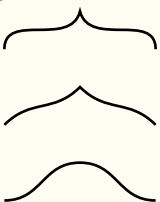
```
(
  fill: none,
  stroke: auto,
  radius: 0.5,
  label-radius: 50%,
  mark: (
    scale: 1,
    length: 0.2,
    width: 0.15,
    inset: 0.05,
    sep: 0.1,
    z-up: (0, 1, 0),
    start: none,
    end: none,
    stroke: auto,
    fill: none,
  ),
)
```

**5.6 Decorations**

Various pre-made shapes and lines.

**5.6.1 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 or none,
  ..style: style
)
```

**start** coordinate  
Start point

**end** coordinate  
End point

**flip** bool  
Flip the brace around

Default: "false"

**debug**

Default: "false"


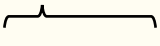
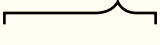
**name** string or none  
Element name used for querying anchors

Default: "none"

**..style** style  
Style key-value pairs

### 5.6.2 flat-brace

Draw a flat curly brace between two points.

|                                                                                     |                                                                                                                       |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
|  | <code>cetz.decorations.flat-brace((0,1),(2,1))</code>                                                                 |
|  | <code>cetz.decorations.flat-brace((0,0),(2,0),</code><br><code>curves: .2,</code><br><code>aspect: 25%)</code>        |
|  | <code>cetz.decorations.flat-brace((0,-1),(2,-1),</code><br><code>outer-curves: 0,</code><br><code>aspect: 75%)</code> |

This mimics the braces from TikZ's `decorations.pathreplacing` library<sup>1</sup>. 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:**

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



|                                                                                   |                            |
|-----------------------------------------------------------------------------------|----------------------------|
| <b>amplitude</b> <code>number</code>                                              | Default: <code>0.3</code>  |
| Determines how much the brace rises above the base line.                          |                            |
| <b>aspect</b> <code>ratio</code>                                                  | Default: <code>50%</code>  |
| Determines the fraction of the total length where the spike will be placed.       |                            |
| <b>curves</b> <code>number</code>                                                 | Default: <code>auto</code> |
| Curviness factor of the brace, a factor of 0 means no curves.                     |                            |
| <b>outer-curves</b> <code>auto</code> or <code>number</code>                      | Default: <code>auto</code> |
| 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`

Flip the brace around

Default: `"false"`

**debug**

Default: `"false"`

**name** `string` or `none`

Element name for querying anchors

Default: `"none"`

**..style** `style`

Style key-value pairs

**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 a 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`  
Update the context's last position resolved coordinate vectors

Default: `"true"`

### 6.2 Styles

#### 6.2.1 resolve

Resolve the current style root

```
(
  fill: none,
  stroke: lpt + luma(0%),
  radius: 1,
  scale: 1,
  length: 0.2,
  width: 0.15,
  inset: 0.05,
  sep: 0.1,
  z-up: (0, 1, 0),
  start: none,
  end: none,
)

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

## 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"

## 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.

```
(
  root: (fill: none, stroke: lpt + luma(0%), radius: 1),
  mark: (
    scale: 1,
    length: 0.2,
    width: 0.15,
    inset: 0.05,
    sep: 0.1,
    z-up: (0, 1, 0),
    start: none,
    end: none,
    stroke: auto,
    fill: none,
  ),
  group: (padding: none),
  line: (
    mark: (
      scale: 1,
      length: 0.2,
      width: 0.15,
      inset: 0.05,
      sep: 0.1,
      z-up: (0, 1, 0),
      start: none,
      end: none,
      stroke: auto,
      fill: none,
    ),
    bezier: (
      mark: (
        scale: 1,
        length: 0.2,
        width: 0.15,
        inset: 0.05,
        sep: 0.1,
        z-up: (0, 1, 0),
        start: none,
        end: none,
        stroke: auto,
        fill: none,
        flex: true,
        position-samples: 30,
      ),
      shorten: "LINEAR",
    ),
    catmull: (
      tension: 0.5,
    ),
  ),
)
```

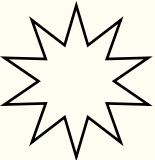
```

mark: (
  scale: 1,
  length: 0.2,
  width: 0.15,
  inset: 0.05,
  sep: 0.1,
  z-up: (0, 1, 0),
  start: none,
  end: none,
  stroke: auto,
  fill: none,
  flex: true,
  position-samples: 30,
),
shorten: "LINEAR",
),
hobby: (
  omega: (1, 1),
  rho: auto,
  mark: (
    scale: 1,
    length: 0.2,
    width: 0.15,
    inset: 0.05,
    sep: 0.1,
    z-up: (0, 1, 0),
    start: none,
    end: none,
    stroke: auto,
    fill: none,
    flex: true,
    position-samples: 30,
  ),
  shorten: "LINEAR",
),
arc: (
  mode: "OPEN",
  mark: (
    scale: 1,
    length: 0.2,
    width: 0.15,
    inset: 0.05,
    sep: 0.1,
    z-up: (0, 1, 0),
    start: none,
    end: none,
    stroke: auto,
    fill: none,
  ),
),
content: (padding: 0, 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, 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 object. Elements or other draw calls may return a modified context element to the canvas to change its state, e.g. modifying the transforming matrix, adding a group or setting a style.

```
(
  typst-style: ..,
  length: 28.35pt,
  debug: false,
  prev: (pt: (0, 0, 0)),
  em-size: (width: 8.8pt, height: 8.8pt),
  style: (:),
  transform: (
    (1, 0, 0.5, 0),
    (0, -1, -0.5, 0),
    (0, 0, 1, 0),
    (0, 0, 0, 1),
  ),
  nodes: (:),
  groups: (),
)

// Show the current context
get-ctx(ctx => {
  content((), raw(repr(ctx), lang: "typc"))
})
```

### 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 object and must return an dictionary of the following keys:

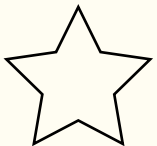
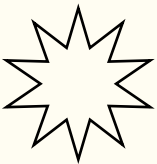
- ctx (required): The (modified) canvas context object
- drawables: List of drawables to render to the canvas
- anchors: A function of the form (<anchor-identifier>) => <vector>
- name: The elements name

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 custom element `my-star` from Section 7:



```
import cetz.drawables: path
import cetz.vectors

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

    // 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, 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.drawables.path((cetz.path-util.line-segment(corners)),
      stroke: style.stroke, fill: style.fill, close: true)

    (ctx: ctx,
     drawables: cetz.drawables.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.