CeTZ

ein Typst Zeichenpacket

Johannes Wolf
fenjalien

Version 0.2.0

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 Zeichenpacket" (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.

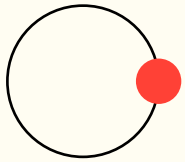
`coordinate` Any coordinate system. See coordinate-systems.

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

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

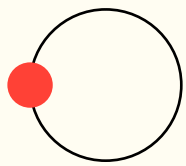
2.2 Anchors

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.



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

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

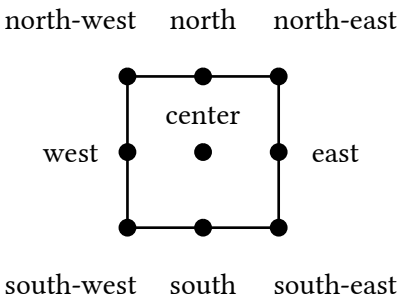
Elements can be placed relative to their own anchors if they have an argument called `anchor`:



```
// An element does not have to be named
// in order to use its own anchors.
circle((0,0), anchor: "west")

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

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:

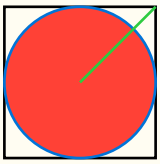
How to fill the drawn element.

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

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

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

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

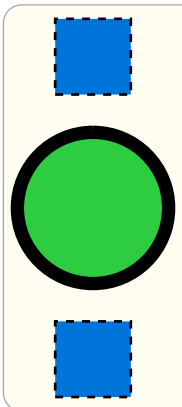


```
// Draws an empty square with a black border
rect((-1, -1), (1, 1))

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

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

When using a dictionary for a style, it is important to note that they update each other instead of overriding the entire option like a non-dictionary value would do. For example, if the stroke is set to `(paint: red, thickness: 5pt)` and you pass `(paint: blue)`, the stroke would become `(paint: blue, thickness: 5pt)`.

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

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



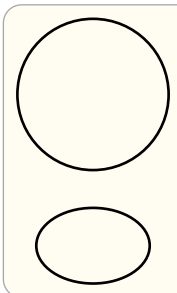
```
set-style(
  // Global fill and stroke
  fill: green,
  stroke: (thickness: 5pt),
  // Stroke and fill for only rectangles
  rect: (stroke: (dash: "dashed"), fill: blue),
)
rect((0,0), (1,1))
circle((0.5, -1.5))
rect((0,-3), (1, -4), stroke: (thickness: 1pt))
```

```
// Its a nice drawing okay
set-style(
  rect: (
    fill: red,
    stroke: none
  ),
  line: (
    fill: blue,
    stroke: (dash: "dashed")
  ),
)
rect((0,0), (1,1))

line((0, -1.5), (0.5, -0.5), (1, -1.5), close: true)

circle((0.5, -2.5), radius: 0.5, fill: green)
```

3.3 Shapes

3.3.1 circle

Draws a circle or ellipse.



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

Parameters

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

position `coordinate`

The position to place the circle on.

Style Root `circle`

Style Keys

radius `number` or `array` Default: 1

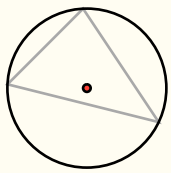
A number that defines the size of the circle's radius. Can also be set to a tuple of two numbers to define the radii of an ellipse, the first number is the x radius and the second is the y radius.

Anchors

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

3.3.2 circle-through

Draws a circle through three coordinates



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

Parameters

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

a `coordinate`
Coordinate a

b `coordinate`
Coordinate b

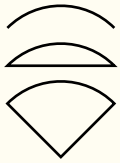
c `coordinate`
Coordinate c

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.



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

Note that two of the three angle arugments (`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 eliptical 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.

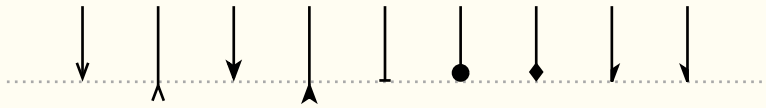
arc-end The position of the arc’s curve end.

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

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

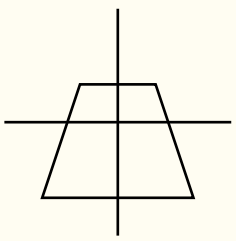
flex `boolean` Default: `true`
Only applicable when marks are used on curves such as bezier and hobby. If true, the mark will point along the secant of the curve. If false, the tangent at the marks tip is used.

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.



```
line((-1.5, 0), (1.5, 0))
line((0, -1.5), (0, 1.5))
line((-1, -1), (-0.5, 0.5), (0.5, 0.5), (1, -1), close: true)
```

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

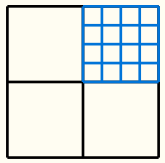
Anchors

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`:


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


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

Parameters

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

..args-style `coordinate` or `content` or `style`

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

angle `angle` or `coordinate` Default: `"0deg"`

Rotates the content by the given angle. A coordinate can be given to rotate the content by the angle between it and the first coordinate given in `args`. This effectively points the right hand side of the content towards the coordinate. This currently exists because Typst's rotate function does not change the width and height of content.

Style Root `content`

Style Keys

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

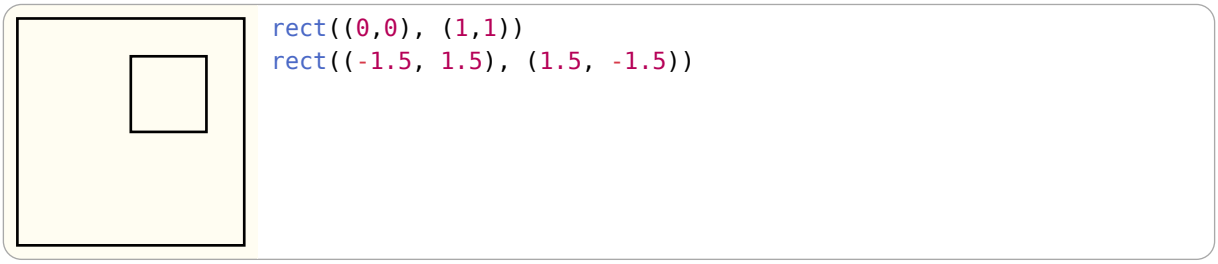
Sets the spacing around content. Can be a single number to set padding on all sides or a dictionary to specify each side specifically. The dictionary follows Typst's `pad` function: https://typst.app/docs/reference/layout/pad/

frame `string` or `none` Default:

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.



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

Style Root `rect`

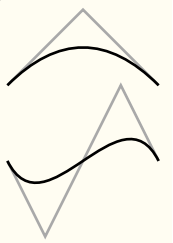
Anchors
Supports compass anchors.

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 rectanlge. 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



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

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

Parameters

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

start `coordinate`
: Start position

end `coordinate`
: End position (last coordinate)

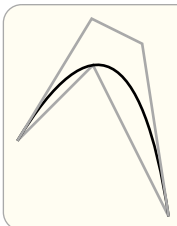
..ctrl-style `coordinate` or `style`
: The first two positional arguments are taken as cubic bezier control points, where the first is the start control point and the second is the end control point. One control point can be given for a quadratic bezier curve instead. Named arguments are for styling.

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`
)

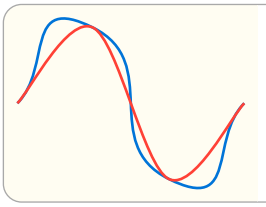
start `coordinate`
: Start position

pass-through `coordinate`
: Curve mid-point

end `coordinate`
End coordinate

3.3.11 catmull

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



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

Parameters

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

..pts-style `coordinate` or `style`
Positional arguments should be coordinates that the curve should pass through. Named arguments are for styling.

close `bool` Default: `"false"`
Closes the curve with a straight line between the start and end of the curve.

Style Root `catmull`
Style Keys

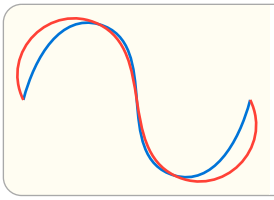
tension `float` Default: `0.5`
I need a description

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.



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

Parameters

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

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

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

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

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

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

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

close `bool` Default: `"false"`

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

Style Root `hobby`

Style Keys

Supports marks.

omega `idk` Default:

The curve's curlyness

rho `idk` Default:

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

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

Parameters

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

body `elements`

Elements with paths to be merged together.

close `bool` Default: `"false"`

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

Anchors

start The start of the merged path.

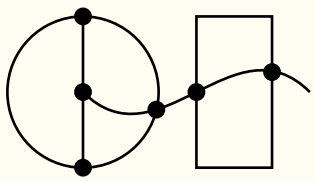
end The end of the merged path.

3.4 Grouping

3.4.1 intersections

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

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



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

Parameters

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

name `string`

Name to prepend to the generated anchors.

body `elements`

Elements to calculate intersections with.

samples `int` Default: `"10"`

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

3.4.2 group

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

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

Parameters

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

body `elements` or `function`

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

Style Root `group`

Style Keys

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

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)`.

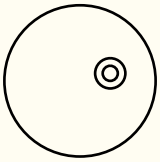
Anchors Supports compass anchors. These are created based on the axis aligned bounding box of all the child elements of the group.

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 overidden 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

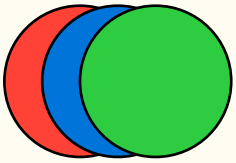
name Default: `"auto"`

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

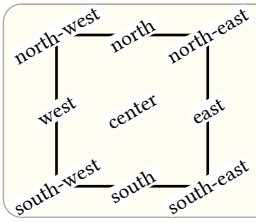
get-ctx(callback: `function`)

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.


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

Parameters

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

layer `float` or `integer`

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

body `elements`

Elements to draw on the layer specified.

3.4.10 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` Default: `"none"`

Element name

3.5 Transformations

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

Transformation matrices get multiplied in the following order:

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

3.5.1 set-transform

Sets the transformation matrix.

Parameters

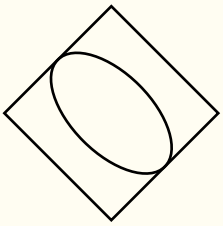
set-transform(mat: `none` `matrix`)

mat `none` or `matrix`

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

3.5.2 rotate

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



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

Parameters

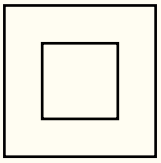
rotate(..angles: `angle`)

..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.



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

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.

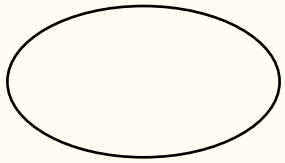
pre `bool` Default: `"true"`

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).



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

Parameters

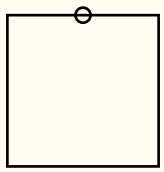
scale(factor: `float` `dictionary`)

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



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

Parameters

set-origin(origin: `coordinate`)

origin `coordinate`

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

3.5.6 move-to

Sets the previous coordinate.

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

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

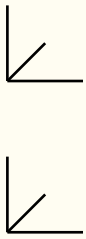
Defines a point `x` units right, `y` units upward, and `z` units away.

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.

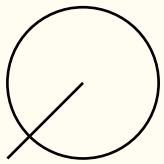
The implicit form can be given as an array of two or three `number` or `length`, as in `(x,y)` and `(x,y,z)`.



```
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

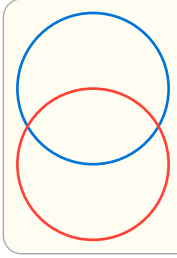
Places the given coordinate relative to the previous coordinate. Or in other words, for the given coordinate, the previous coordinate will be used as the origin. Another coordinate can be given to act as the previous coordinate instead.

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

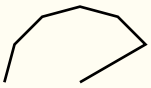
Defines a point a `radius` distance away from the origin at the given `angle`.

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))
```

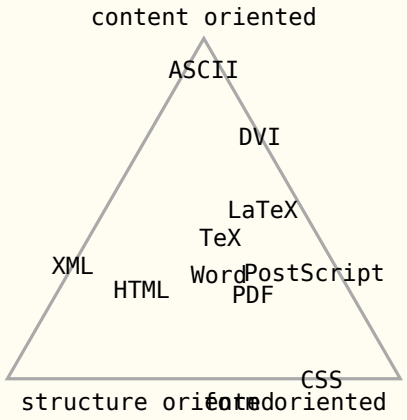
4.5 Barycentric

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 ..., v_n and numbers α_1, α_2, ..., α_n. Then the barycentric coordinate specified by these vectors and numbers is

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

bary `dictionary`

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




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

for (c, a) in (
  ("content", "south"),
  ("structure", "north-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.

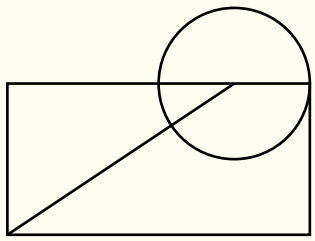
name `string`

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"`.



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

4.7 Tangent

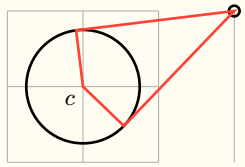
This system allows you to compute the point that lies tangent to a shape. In detail, consider an element and a point. Now draw a straight line from the point so that it "touches" the element (more formally, so that it is *tangent* to this element). The point where the line touches the shape is the point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

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

point `coordinate`
: The point through which the tangent should go.

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

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




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

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

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

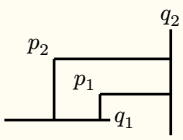
4.8 Perpendicular

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

horizontal `coordinate`
: The coordinate through which the horizontal line passes.

vertical `coordinate`
: The coordinate through which the vertical line passes.

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




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

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

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

4.9 Interpolation

Use this to linearly interpolate between two coordinates `a` and `b` with a given 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.

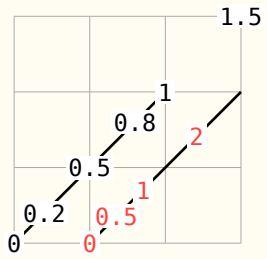
b `coordinate`
The coordinate to interpolate to.

number `number` or `length`
The factor to interpolate by or the distance away from `a` towards `b`.

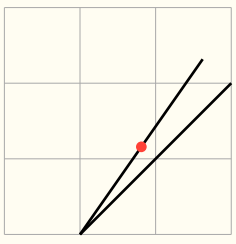
angle `angle` (default: `0deg`)

abs `bool` (default: `false`)
Interpret `number` as absolute distance, instead of a factor.

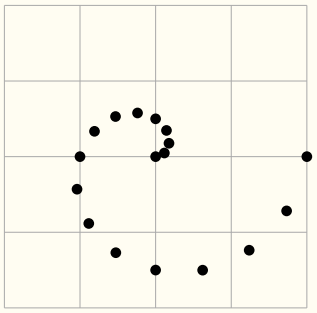
Can be used implicitly as an array in the form `(a, number, b)` or `(a, number, angle, b)`.




```
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])))
}
```

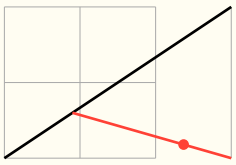


```
grid((0,0), (3,3), help-lines: true)
line((1,0), (3,2))
line((1,0), ((1, 0), 1, 10deg, (3,2)))
fill(red)
stroke(none)
circle(((1, 0), 0.5, 10deg, (3, 2)), radius: 2pt)
```

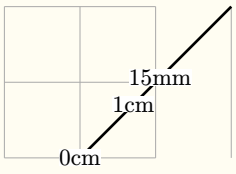


```
grid((0,0), (4,4), help-lines: true)

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

You can even chain them together!



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

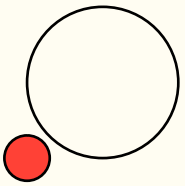



```
grid((0,0), (3, 2), help-lines: true)
line((1,0), (3,2))
for (l, c) in ((0cm, "0cm"), (1cm, "1cm"), (15mm, "15mm")) {
  content(((1,0), l, (3,2)), box(fill: white, $ #c $))
}
```

4.10 Function

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

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



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

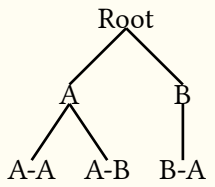
5 Libraries

5.1 Tree

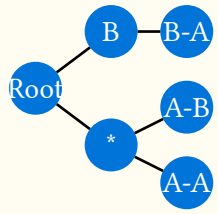
The tree library allows the drawing diagrams with simple tree layout algorithms

5.1.1 tree

Lays out and renders tree nodes.




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




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

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.

draw-edge `auto` or `function` Default: `"auto"`

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

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

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

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

Positioning of parent nodes (begin, center, end)

grow `float` Default: `"1"`

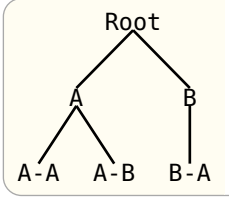
Depth grow factor (default 1)

spread `float` Default: `"1"`

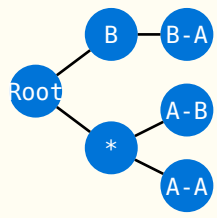
Sibling spread factor (default 1)

name Default: `"none"`

..style




```
import cetz.tree

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




```
import cetz.tree

let data = ([Root], ([\*], [A-A], [A-B]), ([B], [B-A]))
tree.tree(data, content: (padding: .1), direction: "right",
          mark: (end: ">", fill: none),
          draw-node: (node, ..) => {
            circle((), radius: .35, fill: blue, stroke: none)
            content((), text(white, [#node.content]))
          },
          draw-edge: (from, to, ..) => {
            let (a, b) = (from + ".center",
                          to + ".center")

             line((a: a, b: b, abs: true, number: .35),
                  (a: b, b: a, abs: true, number: .35))
          })
```

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 itselfes can be of type `content` or `dictionary` with a key `content`.

5.2 Plot

The library `plot` of CeTZ allows plotting 2D data.

5.2.1 Types

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

`domain` Tuple representing a functions domain as closed interval. Example domains are: `(0, 1)` for $[0, 1]$ or `(-calc.pi, calc.pi)` for $[-\pi, \pi]$.

5.2.2 add-anchor

Add an anchor to a plot environment

Parameters

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

name `string`

Anchor name

position `array`

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

axes `array`

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

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

5.2.3 plot

Create a plot environment

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

Note that different axis-styles can show different axes. The `"school-book"` and `"left"` style shows only axis “x” and “y”, while the `"scientific"` style can show “x2” and “y2”, if set (if unset, “x2” mirrors “x” and “y2” mirrors “y”). Other axes (e.G. “my-axis”) work, but no ticks or labels will be shown.

Options

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

- label (content): Axis label
- min (int): Axis minimum value
- max (int): Axis maximum value
- tick-step (none, float): Distance between major ticks (or no ticks if none)
- minor-tick-step (none, float): Distance between minor ticks (or no ticks if none)
- ticks (array): List of ticks values or value/label tuples. Example `(1,2,3)` or `((1, [A]), (2, [B]),)`
- format (string): Tick label format, `"float"`, `"sci"` (scientific) or a custom function that receives a value and returns a content (`value => content`).
- grid (bool,string): Enable grid-lines at tick values:
  - `"major"`: Enable major tick grid
  - `"minor"`: Enable minor tick grid
  - `"both"`: Enable major & minor tick grid
  - `false`: Disable grid
- unit (none, content): Tick label suffix
- decimals (int): Number of decimals digits to display for tick labels

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

size `array`

Plot canvas size tuple of width and height in canvas units

Default: `(1, 1)`

axis-style `none` or `string`

Axis style “scientific”, “left”, “school-book”
- `"scientific"`: Frame plot area and draw axes y, x, y2, and x2 around it
- `"school-book"`: Draw axes x and y as arrows with both crossing at $(0, 0)$
- `"left"`: Draw axes x and y as arrows, the y axis stays on the left (at `x.min`) and the x axis at the bottom (at `y.min`)
- `none`: Draw no axes (and no ticks).

Default: `"scientific"`

name `string`

Element name

Default: `none`

plot-style `style` or `function`

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

Default: `default-plot-style`

mark-style `style` or `function`

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

Default: `default-mark-style`

fill-below `bool`

Fill functions below the axes (draw axes above fills)

Default: `true`

legend `none` or `auto` or `coordinate`

Position to place the legend at. 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 is read from the legend style (root `legend`).

Default: `auto`

legend-anchor `auto` or `string`

Anchor of the legend group to use as origin of the legend group.

Default: `auto`

legend-style `style`

Legend style orverwrites.

Default: `(:)`

..options `any`

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

- min (int): Axis minimum
- max (int): Axis maximum
- horizontal (bool): Axis orientation; note that each plot must use one vertical and one horizontal axis! The default value for this parameter is guessed: Axes starting with "x" are considered horizontal by default. This does not affect the side the ticks of the axis are drawn, but only the drawing direction.
- tick-step (float): Major tick step
- minor-tick-step (float): Major tick step
- ticks (array): List of ticks values or value/label tuples
- unit (content): Tick label suffix
- decimals (int): Number of decimals digits to display

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: array,
  label,
  data: array function
)
```

domain `domain`

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

Default: `auto`

hypograph `bool`

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

Default: `false`

epigraph `bool`

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

Default: `false`

fill `bool`

Fill to y zero

Default: `false`

fill-type `string`

Fill type:

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

Default: `"axis"`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

mark `string`

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

The following marks are supported:
- `"*"` or `"x"` – X
- `"+"` – Cross
- `"|"` – Bar
- `"-"` – Dash
- `"o"` – Circle
- `"triangle"` – Triangle
- `"square"` – Square

Default: `none`

mark-size `float`

Mark size in cavas units

Default: `.2`

mark-style

Default: `(:)`

samples `int`

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

Default: `50`

sample-at `array`

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

Default: `()`

line `string` or `dictionary`

Line type to use. The following types are supported:
- **`"linear"`** Linear line segments
- **`"spline"`** A smoothed line
- **`"vh"`** Move vertical and then horizontal
- **`"hv"`** Move horizontal and then vertical
- **`"vhv"`** Add a vertical step in the middle
- **`"raw"`** Like linear, but without linearization.

`"linear"` *should* never look different than `"raw"`.

If the value is a dictionary, the type must be supplied via the `type` key. The following extra attributes are supported:
- **`"samples" <int>`** Samples of splines
- **`"tension" <float>`** Tension of splines
- **`"mid" <float>`** Mid-Point of vhv lines (0 to 1)
- **`"epsilon" <float>`** Linearization slope epsilon for use with `"linear"`, defaults to 0.

Default: `"linear"`

axes `array`

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

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

label

Default: `none`

data `array` or `function`

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

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

5.2.5 add-fill-between

Fill the area between two graphs. This behaves same as `plot` but takes a pair of data instead of a single data array/function. The area between both function plots gets filled.

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,
  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`

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

Default: `auto`

samples `int`

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.

Default: `50`

sample-at `array`

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

Default: `()`

line `string` or `dictionary`

Line type to use, see `add`

Default: `"linear"`

axes `array`

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

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

label

Default: `none`

style `style`

Style to use, can be used with a palette function

Default: `(:)`

5.2.6 add-hline

Add horizontal lines at values y

Parameters

```
add-hline(
  ..y: number,
  axes: array,
  style: style,
  label
)
```

..y `number`

Y axis value(s) to add a line at

axes `array`

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

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

style `style`

Style to use, can be used with a palette function

Default: `(:)`

label

Default: `none`

5.2.7 add-vline

Add vertical lines at values x.

Parameters

```
add-vline(
  ..x: number,
  axes: array,
  style: style,
  label
)
```

..x `number`

X axis values to add a line at

axes `array`

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

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

style `style`

Style to use, can be used with a palette function

Default: `(:)`

label

Default: `none`

- add-contour()

5.2.8 add-contour

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

Parameters

```
add-contour(
  data: array function,
  label,
  z: float array,
  x-domain: domain,
  y-domain: domain,
  x-samples: int,
  y-samples: int,
  interpolate: bool,
  op: auto string function,
  axes: array,
  style: style,
  fill: bool,
  limit: int
)
```

data `array` or `function`

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

Examples:
- `(x, y) => x > 0`
- `(x, y) => 30 - (calc.pow(1 - x, 2)+calc.pow(1 - y, 2))`

label

Default: `none`

z `float` or `array`

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

Default: `(1,)`

x-domain `domain`

X axis domain used if `data` is a function.

Default: `(0, 1)`

y-domain `domain`

Y axis domain used if `data` is a function.

Default: `(0, 1)`

x-samples `int`

X axis domain samples (2 < n)

Default: `25`

y-samples `int`

Y axis domain samples (2 < n)

Default: `25`

interpolate `bool`

Use linear interpolation between sample values

Default: `true`

op `auto` or `string` or `function`

Z value comparison oparator:

`">", ">=", "<", "<=", "!=", "=="` Use the operator for comparison.

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

`function` Call comparison function of the format `(plot-z, data-z) => boolean`, where `plot-z` is the z-value from the plots `z` argument and `data-z` is the z-value of the data getting plotted.

Default: `auto`

axes `array`

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

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

style `style`

Style to use, can be used with a palette function

Default: `(:)`

fill `bool`

Fill each contour

Default: `false`

limit `int`

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

Default: `50`

- add-boxwhisker()

5.2.9 add-boxwhisker

Add one or more box or whisker plots

Parameters

```
add-boxwhisker(
  data: array dictionary,
  label,
  axes: array,
  style: style,
  box-width: float,
  whisker-width: float,
  mark: string,
  mark-size: float
)
```

data `array` or `dictionary`

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

The following fields are supported:
- `x` (number) X-axis value
- `min` (number) Minimum value
- `max` (number) Maximum value
- `q1`, `q2`, `q3` (number) Quartiles from lower to to upper
- `outliers` (array of numbers) Optional outliers

Examples:
- ```
  (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
  ```

label

Default: `none`

axes `array`

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

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

style `style`

Style to use, can be used with a palette function

Default: `(:)`

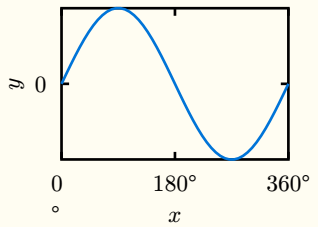
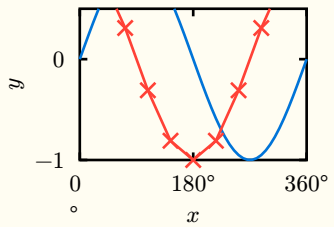
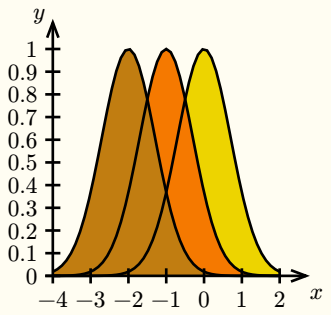
box-width `float`

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

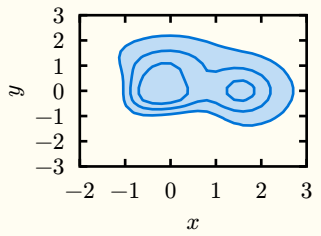
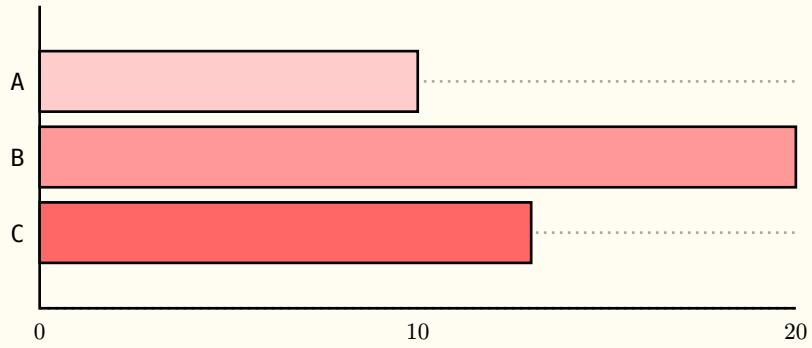
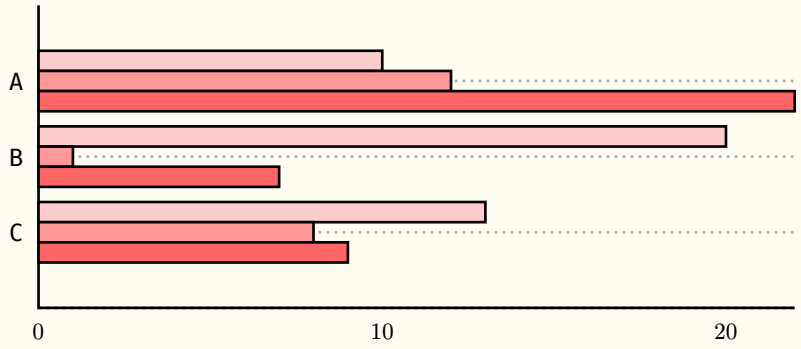
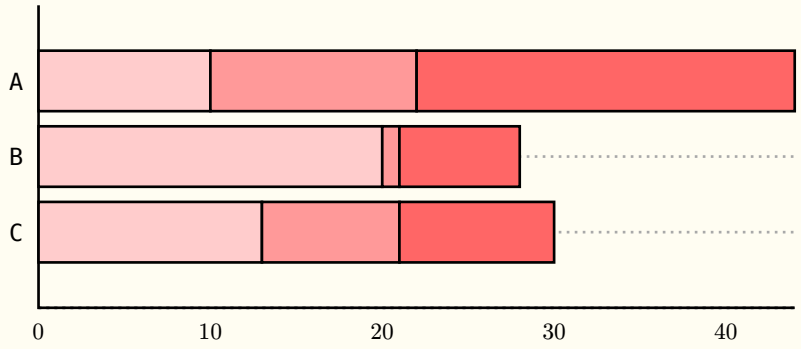
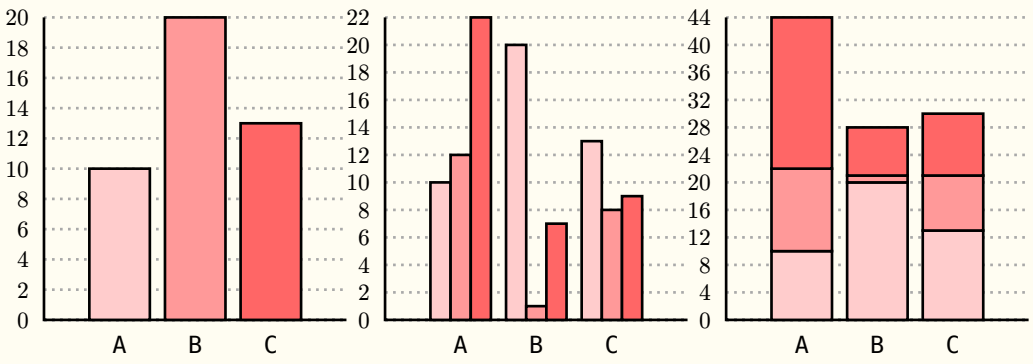
Default: `0.75`

whisker-width `float`

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

Default: `0.5`
  ```

mark `string`

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

Default: `"*"`

mark-size `float`

Size of marks for plotting outliers. Defaults to 0.15

Default: `0.15`

- sample-fn()
- sample-fn2()

5.2.10 sample-fn

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

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

Parameters

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

fn `function`

Function to sample of the form `(x) => y` or `(x) => (x, y)`.

domain `domain`

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

samples `int`

Number of samples in domain.

sample-at `array`

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

Default: `()`

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




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




```
import cetz.plot
import cetz.palette

// Axes can be styled!
// Set the tick length to .1:
set-style(axes: (tick: (length: .1)))

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




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

5.2.13 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 Examples – Bar Chart

Basic




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

Clustered




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

Stacked




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

5.3.2 Examples – Column Chart

Basic, Clustered and Stacked




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

- boxwhisker()

5.3.3 boxwhisker

Add one or more box or whisker plots.

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`

Index in the array where labels of each entry is stored

Default: `0`

box-width `float`

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

Default: `0.75`

whisker-width `float`

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

Default: `0.5`
  ```

mark `string`

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

Default: `"*"`

mark-size `float`

Size of marks for plotting outliers. Defaults to 0.15

Default: `0.15`

..arguments `any`

Additional arguments are passed to `plot.plot`

5.3.4 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.

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`

Style Root: `angle`

Anchors

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

You can use the `radius` and `label-radius` style-keys to set the angle and label radius.

Parameters

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

origin `coordinate`

Angle origin

a `coordinate`

Coordinate of side a

b `coordinate`

Coordinate of side b

inner `bool`

Draw the smaller (inner) angle

Default: `true`

label `none` or `content` or `function`

Draw a label at the angles “label” anchor. If label is a function, it gets the angle value passed as argument.

- name: (none,string): Element value

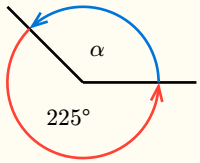
Default: `none`

name

Default: `none`

..style `style`

Style




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

Default `angle` Style

```
(
  fill: none,
  stroke: auto,
  radius: 0.5,
  label-radius: 0.25,
  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,
  ),
)
```