# CeTZ Plot

Johannes Wolf
fenjalien

Version 0.1.3

# 1 Introduction

CeTZ-Plot is a simple plotting library for use with CeTZ.

# 2 Usage

This is the minimal starting point:

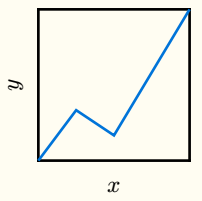
```
#import "@preview/cetz:0.4.2"
#import "@preview/cetz-plot:0.1.3"
#cetz.canvas({
  import cetz.draw: *
  import cetz-plot: *
  ...
})
```

Note that plot functions are imported inside the scope of the `canvas` block. All following example code is expected to be inside a `canvas` block, with the `plot` module imported into the namespace.

# 3 Plot

## 3.1 plot

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




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

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

### 3.1.1 Options

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

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

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

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

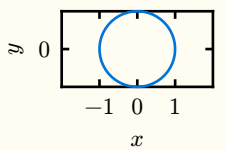
Axis lower domain value. If this is set greater than than `max`, the axis' direction is swapped

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

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

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

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




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

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

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

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

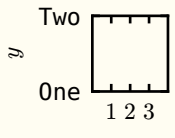
The increment between tick marks on the axis. If set to `auto`, an increment is determined. When set to `none`, incrementing tick marks are disabled.

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

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

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

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




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

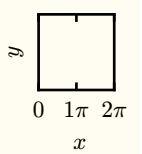
Examples: `(1, 2, 3)` or `((1, [One]), (2, [Two]), (3, [Three]))`

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

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

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

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




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

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

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

**mode:** `none` or `string` Default: `"none"`

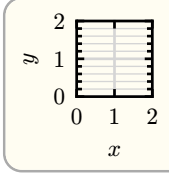
The scaling function of the axis. Takes `lin` (default) for linear scaling, and `log` for logarithmic scaling.

**base:** `none` or `number` Default: `"none"`

The base to be used when labeling axis ticks in logarithmic scaling

**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.




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

**break:** `bool` Default: `"false"`

If true, add a "sawtooth" at the start or end of the axis line, depending on the axis bounds. If the axis min. value is > 0, a sawtooth is added to the start of the axes, if the axis max. value is < 0, a sawtooth is added to its end.

- body (body): Calls of `plot.add` or `plot.add-*` commands. Note that normal drawing commands like `line` or `rect` are not allowed inside the plots body, instead wrap them in `plot.annotate`, which lets you select the axes used for drawing.
- size (array): Plot size tuple of `(<width>, <height>)` in canvas units. This is the plots inner plotting size without axes and labels.
- axis-style (none, string): How the axes should be styled:

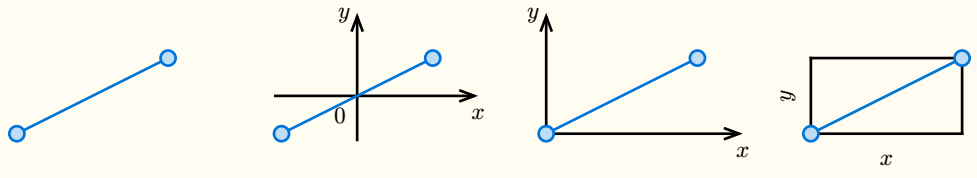
  **scientific** Frames plot area using a rectangle and draw axes `x` (bottom), `y` (left), `x2` (top), and `y2` (right) around it. If `x2` or `y2` are unset, they mirror their opposing axis.

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

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

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

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




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

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

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

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

Stroke style to use for stroking the graph.

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

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

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

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

Stroke style to use for stroking the mark.

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

Paint to use for filling marks.

- fill-below (bool): If true, the filled shape of plots is drawn *below* axes.
- name (string): The plots element name to be used when referring to anchors
- legend (none, auto, coordinate): The position the legend will be drawn at. See plot-legends for information about legends. If set to `<auto>`, the legend's "default-placement" styling will be used. If set to a `<coordinate>`, it will be taken as relative to the plot's origin.
- legend-anchor (auto, string): Anchor of the legend group to use as its origin. If set to `auto` and `lengend` is one of the predefined legend anchors, the opposite anchor to `legend` gets used.
- legend-style (style): Style key-value overwrites for the legend style with style root `legend`.
- ..options (any): Axis options, see *options* below.

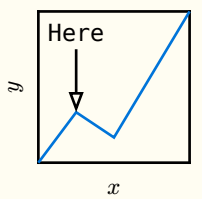
### 3.1.2 Parameters

```
plot(
  body,
  size,
  axis-style,
  name,
  plot-style,
  mark-style,
  fill-below,
  legend,
  legend-anchor,
  legend-style,
  ..options
)
```

## 3.2 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.




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

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

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

#### 3.2.1 Parameters

```
add-anchor(
  name,
  position,
  axes
)
```

### 3.3 add

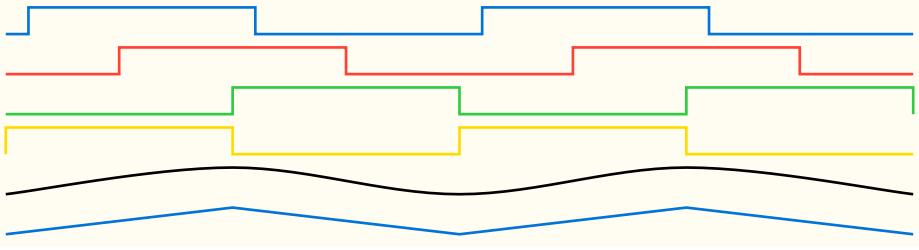
Add data to a plot environment.

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.

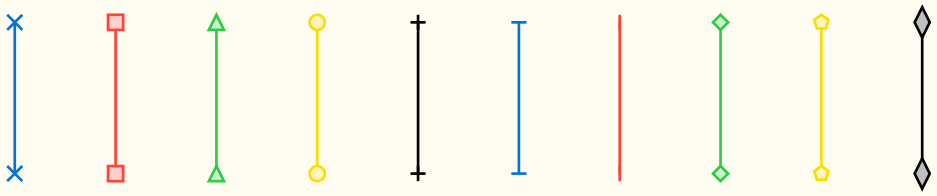
- domain (domain): Domain of `data`, if `data` is a function. Has no effect if `data` is not a function.
- hypograph (bool): Fill hypograph; uses the `hypograph` style key for drawing
- epigraph (bool): Fill epigraph; uses the `epigraph` style key for drawing
- fill (bool): Fill the shape of the plot
- fill-type (string): Fill type:
  **`"axis"`** Fill the shape to y = 0
  **`"shape"`** Fill the complete shape
- samples (int): Number of times the `data` function gets called for sampling y-values. Only used if `data` is of type function. This parameter gets passed onto `sample-fn`.
- sample-at (array): Array of x-values the function gets sampled at in addition to the default sampling. This parameter gets passed to `sample-fn`.
- line (string, dictionary): Line type to use. The following types are supported:
  **`"raw"`** Plot raw data
  **`"linear"`** Linearize data
  **`"spline"`** Calculate a Catmull-Rom curve through all points
  **`"vh"`** Move vertical and then horizontal
  **`"hv"`** Move horizontal and then vertical
  **`"hvh"`** Add a vertical step in the middle

  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.



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

- style (style): Style to use, can be used with a `palette` function
- axes (axes): Name of the axes to use for plotting. Reversing the axes means rotating the plot by 90 degrees.
- mark (string): Mark symbol to place at each distinct value of the graph. Uses the `mark` style key of `style` for drawing:

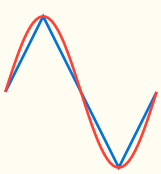


```
let points(offset) = ((offset, 0), (offset, 1))

plot.plot(size: (12, 2), axis-style: none, {
  plot.add(points(0), mark: "*")  // same as "x"
  plot.add(points(1), mark: "square")
  plot.add(points(2), mark: "triangle")
  plot.add(points(3), mark: "o")
  plot.add(points(4), mark: "+")
  plot.add(points(5), mark: "-")
  plot.add(points(6), mark: "|")
  plot.add(points(7), mark: "<>")
  plot.add(points(8), mark: 5)  // specify an integer to use a regular polygon.

  // Alternatively, specify custom function:
  plot.add(points(9), mark: (pt, size, style) => {
    let top = (to: pt, rel: (0, size))
    let bot = (to: pt, rel: (0, -size))
    let left = (to: pt, rel: (-size/2, 0))
    let right = (to: pt, rel: (size/2, 0))
    cetz.draw.line(top, left, bot, right, close: true, ..style)
  })
 })
```

- mark-size (float): Mark size in canvas units
- data (array,function): Array of 2D data points (numeric) or a function of the form `x => y`, where `x` is a value in `domain` and `y` must be numeric or a 2D vector (for parametric functions).



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

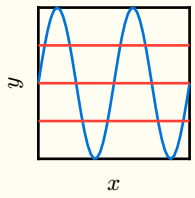
- label (none,content): Legend label to show for this plot.

#### 3.3.1 Parameters

```
add(
  domain,
  hypograph,
  epigraph,
  fill,
  fill-type,
  style,
  mark,
  mark-size,
  mark-style,
  samples,
  sample-at,
  line,
  axes,
  label,
  data
)
```

## 3.4 add-hline

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




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

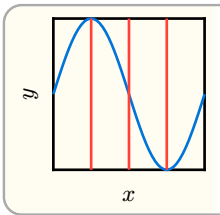
- ..y (float): Y axis value(s) to add a line at
- min (auto,float): X axis minimum value or auto to take the axis minimum
- max (auto,float): X axis maximum value or auto to take the axis maximum
- axes (array): Name of the axes to use for plotting
- style (style): Style to use, can be used with a palette function
- label (none,content): Legend label to show for this plot.

### 3.4.1 Parameters

```
add-hline(
  ..y,
  min,
  max,
  axes,
  style,
  label
)
```

## 3.5 add-vline

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




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

- ..x (float): X axis values to add a line at
- min (auto,float): Y axis minimum value or auto to take the axis minimum
- max (auto,float): Y axis maximum value or auto to take the axis maximum
- axes (array): Name of the axes to use for plotting, note that not all plot styles are able to display a custom axis!
- style (style): Style to use, can be used with a palette function
- label (none,content): Legend label to show for this plot.

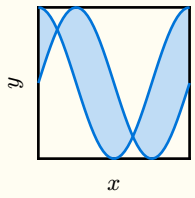
### 3.5.1 Parameters

```
add-vline(
  ..x,
  min,
  max,
  axes,
  style,
  label
)
```

## 3.6 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.




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

- 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.
- 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.
- sample-at (array): Array of x-values the function(s) get sampled at in addition to the default sampling.
- line (string, dictionary): Line type to use, see `add()`.
- style (style): Style to use, can be used with a palette function.
- label (none,content): Legend label to show for this plot.
- axes (array): Name of the axes to use for plotting.
- data-a (array,function): Data of the first plot, see `add()`.
- data-b (array,function): Data of the second plot, see `add()`.

#### 3.6.1 Parameters

```
add-fill-between(
  data-a,
  data-b,
  domain,
  samples,
  sample-at,
  line,
  axes,
  label,
  style
)
```

### 3.7 add-bar

Add a bar- or column-chart to the plot

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

- data (array): Array of data items. An item is an array containing a x an one or more y values. For example `(0, 1)` or `(0, 10, 5, 30)`. Depending on the `mode`, the data items get drawn as either clustered or stacked rects.
- x-key (int,string): Key to use for retrieving a bars x-value from a single data entry. This value gets passed to the `.at(...)` function of a data item.
- y-key (auto,int,string,array): Key to use for retrieving a bars y-value. For clustered/stacked data, this must be set to a list of keys (e.g. `range(1, 4)`). If set to `auto`, att but the first array-values of a data item are used as y-values.
- error-key (none,int,string,array): Key(s) to use for retrieving a bars y-error.
- mode (string): The mode on how to group data items into bars:
  - **basic** Add one bar per data value. If the data contains multiple values, group those bars next to each other.
  - **clustered** Like "basic", but take into account the maximum number of values of all items and group each cluster of bars together having the width of the widest cluster.

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

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

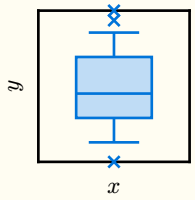
- labels (none,content,array): A single legend label for "basic" bar-charts, or a a list of legend labels per bar category, if the mode is one of "clustered", "stacked" or "stacked100".
- bar-width (float): Width of one data item on the y axis
- bar-position (string): Positioning of data items relative to their x value.
  - "start": The lower edge of the data item is on the x value (left aligned)
  - "center": The data item is centered on the x value
  - "end": The upper edge of the data item is on the x value (right aligned)
- cluster-gap (float): Spacing between bars insides a cluster.
- style (dictionary): Plot style
- axes (axes): Plot axes. To draw a horizontal growing bar chart, you can swap the x and y axes.

### 3.7.1 Parameters

```
add-bar(
  data,
  x-key,
  y-key,
  error-key,
  mode,
  labels,
  bar-width,
  bar-position,
  cluster-gap,
  whisker-size,
  error-style,
  style,
  axes
)
```

## 3.8 add-boxwhisker

Add one or more box or whisker plots




```
plot.plot(size: (2,2), x-tick-step: none, y-tick-step: none, {
  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
})
```

- data (array, dictionary): dictionary or array of dictionaries containing the needed entries to plot box and whisker plot.

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

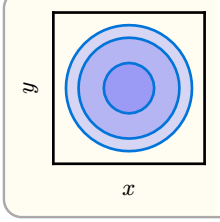
- axes (array): Name of the axes to use (“x”, “y”), note that not all plot styles are able to display a custom axis!
- style (style): Style to use, can be used with a palette function
- box-width (float): Width from edge-to-edge of the box of the box and whisker in plot units. Defaults to 0.75
- whisker-width (float): Width from edge-to-edge of the whisker of the box and whisker in plot units. Defaults to 0.5
- mark (string): Mark to use for plotting outliers. Set `none` to disable. Defaults to “x”
- mark-size (float): Size of marks for plotting outliers. Defaults to 0.15
- label (none,content): Legend label to show for this plot.

#### 3.8.1 Parameters

```
add-boxwhisker(
  data,
  label,
  axes,
  style,
  box-width,
  whisker-width,
  mark,
  mark-size
)
```

### 3.9 add-contour

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




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

- data (array, function): A function of the signature `(x, y) => z` or an array of arrays of floats (a matrix) where the first index is the row and the second index is the column.
- z (float, 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 the order of specification.
- x-domain (domain): X axis domain used if `data` is a function, that is the domain inside the function gets sampled.
- y-domain (domain): Y axis domain used if `data` is a function, see `x-domain`.
- x-samples (int): X axis domain samples (2 < n). Note that contour finding can be quite slow. Using a big sample count can improve accuracy but can also lead to bad compilation performance.
- y-samples (int): Y axis domain samples (2 < n)
- interpolate (bool): Use linear interpolation between sample values which can improve the resulting plot, especially if the contours are curved.
- op (auto,string,function): Z value comparison oparator:

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

  **`auto`** Use “>=” for positive z values, “<=” for negative z values.

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

- fill (bool): Fill each contour
- style (style): Style to use for plotting, can be used with a palette function. Note that all z-levels use the same style!
- axes (axes): Name of the axes to use for plotting.
- limit (int): Limit of contours to create per z value before the function panics
- label (none,content): Plot legend label to show. The legend preview for contour plots is a little rectangle drawn with the contours style.

#### 3.9.1 Parameters

```
add-contour(
  data,
  label,
  z,
  x-domain,
  y-domain,
  x-samples,
  y-samples,
  interpolate,
  op,
  axes,
  style,
  fill,
  limit
)
```

### 3.10 add-errorbar

Add x- and/or y-error bars

- pt (tuple): Error-bar center coordinate tuple: `(x, y)`
- x-error (float,tuple): Single error or tuple of errors along the x-axis
- y-error (float,tuple): Single error or tuple of errors along the y-axis
- mark (none,string): Mark symbol to show at the error position (`pt`).
- mark-size (number): Size of the mark symbol.
- mark-style (style): Extra style to apply to the mark symbol.
- whisker-size (float): Width of the error bar whiskers in canvas units.
- style (dictionary): Style for the error bars
- label (none,content): Label to tsh
- axes (axes): Plot axes. To draw a horizontal growing bar chart, you can swap the x and y axes.

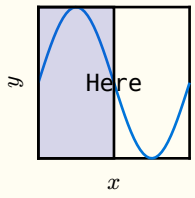
### 3.10.1 Parameters

```
add-errorbar(
  pt,
  x-error,
  y-error,
  label,
  mark,
  mark-size,
  mark-style,
  whisker-size,
  style,
  axes
)
```

## 3.11 annotate

Add an annotation to the plot

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




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

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

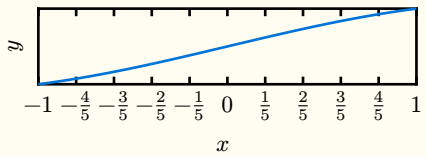
- body (drawable): Elements to draw
- axes (axes): X and Y axis names
- resize (bool): If true, the plots axes get adjusted to contain the annotation
- padding (none,number,dictionary): Annotation padding that is used for axis adjustment
- background (bool): If true, the annotation is drawn behind all plots, in the background. If false, the annotation is drawn above all plots.

### 3.11.1 Parameters

```
annotate(
  body,
  axes,
  resize,
  padding,
  background
)
```

## 3.12 fraction

Fraction tick formatter




```
plot.plot(size: (5,1),
          x-format: plot.formats.fraction,
          x-tick-step: 1/5,
          y-tick-step: none, {
  plot.add(calc.sin, domain: (-1, 1))
})
```

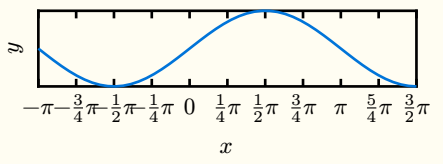
- value (number): Value to format
- denom (auto, int): Denominator for result fractions. If set to `auto`, a hardcoded fraction table is used for finding fractions with a denominator <= 11.
- eps (number): Epsilon used for comparison

#### 3.12.1 Parameters

```
fraction(
  value,
  denom,
  eps
) -> Content if a matching fraction could be found or none
```

## 3.13 multiple-of

Multiple of tick formatter




```
plot.plot(size: (5,1),
          x-format: plot.formats.multiple-of,
          x-tick-step: calc.pi/4,
          y-tick-step: none, {
  plot.add(calc.sin, domain: (-calc.pi, 1.5 * calc.pi))
})
```

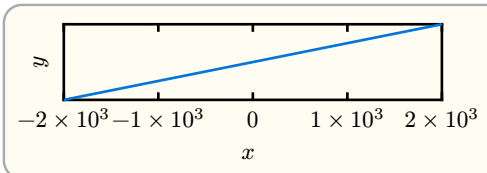
- value (number): Value to format
- factor (number): Factor value is expected to be a multiple of.
- symbol (content): Suffix symbol. For `value` = 0, the symbol is not appended.
- fraction (none, true, int): If not none, try finding matching fractions using the same mechanism as `fraction`. If set to an integer, that integer is used as denominator. If set to `none` or `false`, or if no fraction could be found, a real number with `digits` digits is used.
- digits (int): Number of digits to use for rounding
- eps (number): Epsilon used for comparison
- prefix (content): Content to prefix
- suffix (content): Content to append

#### 3.13.1 Parameters

```
multiple-of(
  value,
  factor,
  symbol,
  fraction,
  digits,
  eps,
  prefix,
  suffix
) -> Content if a matching fraction could be found or none
```

## 3.14 sci

Scientific notation tick formatter




```
plot.plot(size: (5,1),
          x-format: plot.formats.sci,
          x-tick-step: 1e3,
          y-tick-step: none, {
  plot.add(x => x, domain: (-2e3, 2e3))
})
```

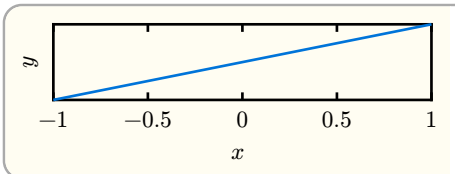
- value (number): Value to format
- digits (int): Number of digits for rounding the factor
- prefix (content): Content to prefix
- suffix (content): Content to append

### 3.14.1 Parameters

```
sci(
  value,
  digits,
  prefix,
  suffix
) -> Content
```

## 3.15 decimal

Rounded decimal number formatter




```
plot.plot(size: (5,1),
          x-format: plot.formats.decimal,
          x-tick-step: .5,
          y-tick-step: none, {
  plot.add(x => x, domain: (-1, 1))
})
```

- value (number): Value to format
- digits (int): Number of digits to round to
- prefix (content): Content to prefix
- suffix (content): Content to append

### 3.15.1 Parameters

```
decimal(
  value,
  digits,
  prefix,
  suffix
) -> Content
```

## 3.16 add-violin

Add a violin plot

A violin plot is a chart that can be used to compare the distribution of continuous data between categories.

- data (array): Array of data items. An item is an array containing an x and one or more y values.
- x-key (int, string): Key to use for retrieving the x position of the violin.
- y-key (int, string): Key to use for retrieving values of points within the category.
- side (string): The sides of the violin to be rendered:

**left** Plot only the left side of the violin.
**right** Plot only the right side of the violin.
**both** Plot both sides of the violin.

- kernel (function): The kernel density estimator function, which takes a single x value relative to the center of a distribution (0) and normalized by the bandwidth
- bandwidth (float): The smoothing parameter of the kernel.
- extents (float): The extension of the domain, expressed as a fraction of spread.
- samples (int): The number of samples of the kernel to render.
- style (dictionary): Style override dictionary.
- mark-style (dictionary): (unused, will eventually be used to render interquartile ranges).
- axes (axes): (unstable, documentation to follow once completed).
- label (none, content): The name of the category to be shown in the legend.

### 3.16.1 Parameters

```
add-violin(
  data,
  x-key,
  y-key,
  side,
  kernel,
  bandwidth,
  extents,
  samples,
  style,
  mark-style,
  axes,
  label
)
```

## 3.17 item

Construct a legend item for use with the `legend` function

- label (none, auto, content): Legend label or auto to use the enumerated default label
- preview (auto, function): Legend preview icon function of the format `item => elements`. Note that the canvas bounds for drawing the preview are (0,0) to (1,1).
- mark (none,string): Legend mark symbol
- mark-style (none,dictionary): Mark style
- mark-size (number): Mark size
- ..style (styles): Style keys for the single item

### 3.17.1 Parameters

```
item(
  label,
  preview,
  mark,
  mark-style,
  mark-size,
  ..style
)
```

## 3.18 legend

Draw a legend

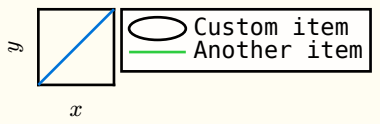
### 3.18.1 Parameters

```
legend(
  position,
  items,
  name,
  ..style
)
```

## 3.19 add-legend

Function for manually adding a legend item from within a plot environment

- label (content): Legend label
- preview (auto,function): Legend preview function of the format `() => elements`. The preview canvas bounds are between (0,0) and (1,1). If set to `auto`, a straight line is drawn.




```
plot.plot(size: (1,1), x-tick-step: none, y-tick-step: none, {
  plot.add(((0,0), (1,1))) // Some data
  plot.add-legend([Custom item], preview: () => {
    import cetz.draw: *
    circle((.5,.5), radius: .5) // Draw a custom preview
                                // between (0,0) and (1,1)
  })
  plot.add-legend([Another item])
})
```

### 3.19.1 Parameters

```
add-legend(
  label,
  preview
)
```

## 3.20 Styling

You can use style root `axes` with the following keys:

## 3.21 default-style

Default axis style

**tick-limit:** `int` Default: `100`

Upper major tick limit.

**minor-tick-limit:** `int` Default: `1000`

Upper minor tick limit.

**auto-tick-factors:** `array` Default: `none`

List of tick factors used for automatic tick step determination.

**auto-tick-count:** `int` Default: `none`

Number of ticks to generate by default.

**stroke:** `stroke` Default: `none`

Axis stroke style.

**label.offset:** `number` Default: `none`

Distance to move axis labels away from the axis.

**label.anchor:** `anchor` Default: `none`

Anchor of the axis label to use for it's placement.

**label.angle:** `angle` Default: `none`

Angle of the axis label.

**axis-layer:** `float` Default: `none`

Layer to draw axes on (see cetz' `on-layer`)

**grid-layer:** `float` Default: `none`

Layer to draw the grid on (see cetz' `on-layer`)

**background-layer:** `float` Default: `none`

Layer to draw the background on (see cetz' `on-layer`)

**padding:** `number` Default: `none`

Extra distance between axes and plotting area. For schoolbook axes, this is the length of how much axes grow out of the plotting area.

**overshoot:** `number` Default: `none`

School-book style axes only: Extra length to add to the end (right, top) of axes.

**tick.stroke:** `stroke` Default: `none`

Major tick stroke style.

**tick.minor-stroke:** `stroke` Default: `none`

Minor tick stroke style.

**tick.offset:** `number` or `ratio` Default: `none`

Major tick offset along the tick's direction, can be relative to the length.

**tick.minor-offset:** `number` or `ratio` Default: `none`

Minor tick offset along the tick's direction, can be relative to the length.

**tick.length:** `number` Default: `none`

Major tick length.

**tick.minor-length:** `number` or `ratio` Default: `none`

Minor tick length, can be relative to the major tick length.

**tick.label.offset:** `number` Default: `none`

Major tick label offset away from the tick.

**tick.label.angle:** `angle` Default: `none`

Major tick label angle.

**tick.label.anchor:** `anchor` Default: `none`

Anchor of major tick labels used for positioning.

**tick.label.show:** `auto` or `bool` Default: `auto`

Set visibility of tick labels. A value of `auto` shows tick labels for all but mirrored axes.

**grid.stroke:** `stroke` Default: `none`

Major grid line stroke style.

**break-point.width:** `number` Default: `none`

Axis break width along the axis.

**break-point.length:** `number` Default: `none`

Axis break length.

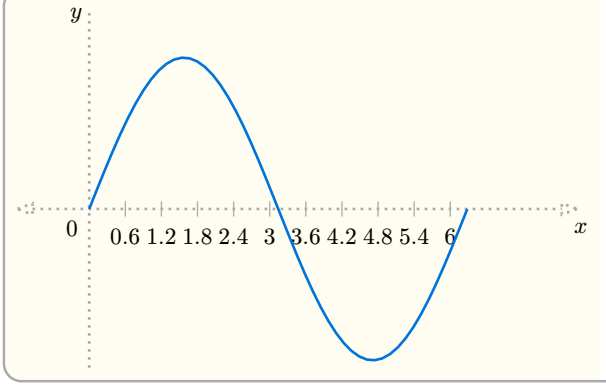
**minor-grid.stroke:** `stroke` Default: `none`

Minor grid line stroke style.

**shared-zero:** `bool` or `content` Default: `"$0$"`

School-book style axes only: Content to display at the plots origin (0,0). If set to `false`, nothing is shown. Having this set, suppresses auto-generated ticks for 0!

#### 3.21.1 Example




```
import cetz.draw: *
import cetz-plot: *

set-style(axes: (
  stroke: (dash: "dotted", paint: gray),
  x: (mark: (start: ">", end: ">"), padding: 1),
  y: (mark: none),
  tick: (stroke: gray + .5pt),
))

plot.plot(size: (5, 4), axis-style: "school-book",
y-tick-step: none, {
  plot.add(calc.sin, domain: (0, calc.pi * 2))
})
```

# 4 Chart

## 4.1 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.

#### 4.1.1 Styling

Can be applied with `cetz.draw.set-style(barchart: (bar-width: 1))`.

**Root**: `barchart`.

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

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

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

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

**cluster-gap:** `float` Default: `0`

Spacing between bars insides a cluster.

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

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

- 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**

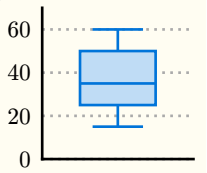
  ```
  (([A], 1), ([B], 2), ([C], 3),)
  ```
- label-key (int,string): Key to access the label of a data row. This key is used as argument to the rows `.at(..)` function.
- value-key (int,string): Key(s) to access values of a data row. These keys are used as argument to the rows `.at(..)` function.
- error-key (none,int,string,array): Key(s) to access error values of a data row. These keys are used as argument to the rows `.at(..)` function.
- mode (string): Chart mode:
  **basic** Single bar per data row
  **clustered** Group of bars per data row
  **stacked** Stacked bars per data row
  **stacked100** Stacked bars per data row relative to the sum of the row
- size (array): Chart size as width and height tuple in canvas unist; width can be set to `auto`.
- bar-style (style,function): Style or function (idx => style) to use for each bar, accepts a palette function.
- y-label (content,none): Y axis label
- x-label (content,none): x axis label
- labels (none,content): Legend labels per x value group
- ..plot-args (any): Arguments to pass to `plot.plot`

### 4.1.2 Parameters

```
barchart(
  data,
  label-key,
  value-key,
  error-key,
  mode,
  size,
  bar-style,
  x-label,
  x-format,
  y-label,
  labels,
  ..plot-args
)
```

## 4.2 boxwhisker

Add one or more box or whisker plots.




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

#### 4.2.1 Styling

**Root** `boxwhisker`

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

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

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

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

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

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

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

- data (array, 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
- label-key (integer, string): Index in the array where labels of each entry is stored
- mark (string): Mark to use for plotting outliers. Set `none` to disable. Defaults to "x"
- ..plot-args (any): Additional arguments are passed to `plot.plot`

#### 4.2.2 Parameters

```
boxwhisker(
  data,
  size,
  label-key,
  mark,
  ..plot-args
)
```

### 4.3 columnchart

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

#### 4.3.1 Styling

**Root**: `columnchart`.

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

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

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

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

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

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

- 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**

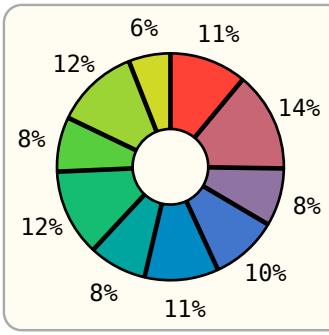
  ```
  (([A], 1), ([B], 2), ([C], 3),)
  ```
- label-key (int,string): Key to access the label of a data row. This key is used as argument to the rows `.at(..)` function.
- value-key (int,string): Key(s) to access value(s) of data row. These keys are used as argument to the rows `.at(..)` function.
- error-key (none,int,string,array): Key(s) to access error values of a data row. These keys are used as argument to the rows `.at(..)` function.
- mode (string): Chart mode:
  **basic** Single bar per data row
  **clustered** Group of bars per data row
  **stacked** Stacked bars per data row
  **stacked100** Stacked bars per data row relative to the sum of the row
- size (array): Chart size as width and height tuple in canvas unist; width can be set to `auto`.
- bar-style (style,function): Style or function (idx => style) to use for each bar, accepts a palette function.
- y-label (content,none): Y axis label
- x-label (content,none): x axis label
- labels (none,content): Legend labels per y value group
- ..plot-args (any): Arguments to pass to `plot.plot`

### 4.3.2 Parameters

```
columnchart(
  data,
  label-key,
  value-key,
  error-key,
  mode,
  size,
  bar-style,
  x-label,
  y-format,
  y-label,
  labels,
  ..plot-args
)
```

## 4.4 piechart

Draw a pie- or donut-chart




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

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

### 4.4.1 Styling

**Root** `piechart`

**radius:** `number` Default: `1`

Outer radius of the chart.

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

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

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

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

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

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

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

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

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

**start:** `angle` Default: `90deg`

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

**stop:** `angle` Default: `450deg`

The pie-charts stop angle (ccw).

**clockwise:** `bool` Default: `true`

The pie-charts rotation direction.

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

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

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

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

**VALUE** Display the slices value

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

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

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

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

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

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

The anchor of the outer label to use for positioning.

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

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

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

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

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

See `outer-label.angle`.

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

See `outer-label.anchor`.

**legend.label:** `none` or `string` or `function` Default: `"LABEL"`

See `outer-label.content`. The legend gets shown if this key is set != none.

### 4.4.2 Anchors

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

- data (array): Array of data items. A data item can be:
  - A number: A number that is used as the fraction of the slice
  - An array: An array which is read depending on value-key, label-key and outset-key
  - A dictionary: A dictionary which is read depending on value-key, label-key and outset-key
- value-key (none,int,string): Key of the "value" of a data item. If for example data items are passed as dictionaries, the value-key is the key of the dictionary to access the items chart value.
- label-key (none,int,string): Same as the value-key but for getting an items label content.
- outset-key (none,int,string): Same as the value-key but for getting if an item should get outset (highlighted). The outset can be a bool, float or ratio. If of type `bool`, the outset distance from the style gets used.
- outset (none,int,array): A single or multiple indices of items that should get offset from the center to the outsides of the chart. Only used if outset-key is none!
- slice-style (function,array,gradient): Slice style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each slice the style at the slices index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the the slices index divided by the number of slices as position on the gradient.

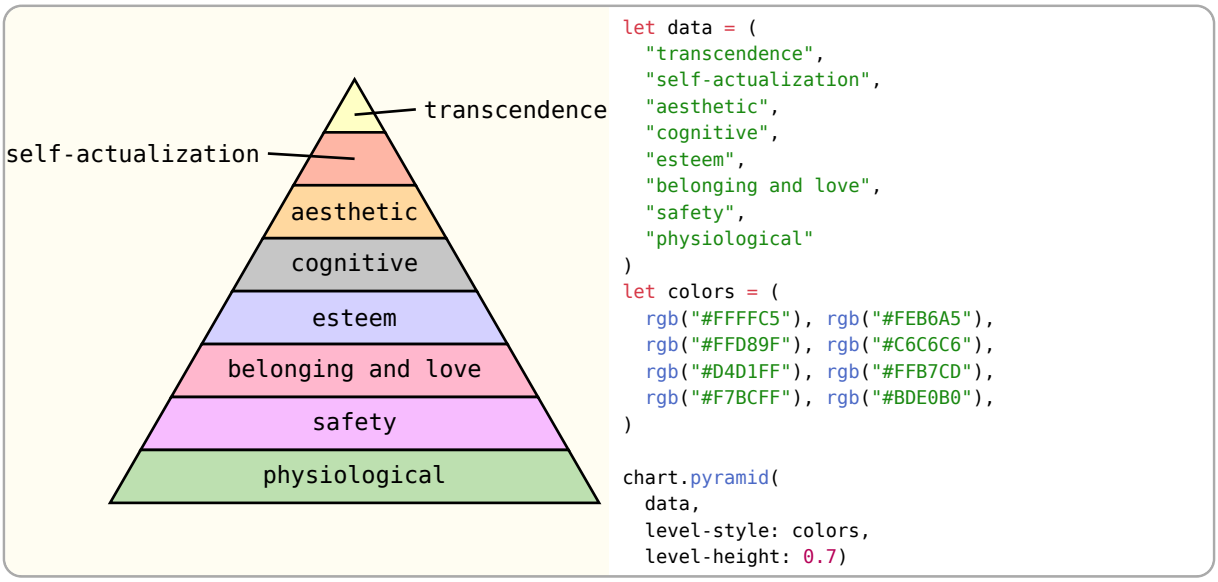
If one of stroke or fill is not in the style dictionary, it is taken from the charts style.

#### 4.4.3 Parameters

```
piechart(
  data,
  value-key,
  label-key,
  outset-key,
  outset,
  slice-style,
  name,
  ..style
)
```

## 4.5 pyramid

Draw a pyramid chart




```
let data = (
  "transcendence",
  "self-actualization",
  "aesthetic",
  "cognitive",
  "esteem",
  "belonging and love",
  "safety",
  "physiological"
)
let colors = (
  rgb("#FFFFC5"), rgb("#FEB6A5"),
  rgb("#FFD89F"), rgb("#C6C6C6"),
  rgb("#D4D1FF"), rgb("#FFB7CD"),
  rgb("#F7BCFF"), rgb("#BDE0B0"),
)

chart.pyramid(
  data,
  level-style: colors,
  level-height: 0.7)
```

#### 4.5.1 Styling

**Root** `pyramid`

**level-height:** `number` Default: `1`

Minimum level height.

**gap:** `number` or `ratio` Default: `0`

Gap between levels to leave empty. If `mode` is "AREA-HEIGHT", the value must be a ratio and will be proportional to the height of the first level.

**mode:** `string` Default: `"REGULAR"`

The mode of how to shape each level:

- "REGULAR": All levels have the same height and make a perfectly triangular pyramid
- "AREA-HEIGHT": The area of each level is proportional to its value. Only the height is adapted, keeping the pyramid triangular
- "HEIGHT": The height of each level is proportional to its value. The pyramid is kept as a perfect triangle
- "WIDTH": The height of each level is fixed, but its width is proportional to the value. The pyramid might not be perfectly triangular

**side-label.content:** `none` or `string` or `function` Default: `none`

Content to display outsides the charts levels, on the side. There are the following predefined values:

**LABEL** Display the levels label (see `label-key`)

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

**VALUE** Display the levels value

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

**side-label.side:** `string` Default: `"west"`

The side of the chart on which to place side labels, either "west" or "east"

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

Content to display insides the charts levels. See `side-label.content` for the possible values.

**inner-label.force-inside:** `boolean` Default: `false`

If false, labels are automatically placed outside their correspoding levels if they don't fit inside. If true, they are always placed inside.

### 4.5.2 Anchors

The chart places one anchor per item at the center of its level that gets named `"levels.<index>"`, one on the middle of its left side named `"levels.<index>.west"`, and one on the right side named `"levels.<index>.east"`, where index is the index of the level data in `data`.

- data (array): Array of data items. A data item can be:
  - A number: A number that is used as the fraction of the level
  - An array: An array which is read depending on value-key and label-key
  - A dictionary: A dictionary which is read depending on value-key and label-key
- value-key (none,int,string): Key of the "value" of a data item. If for example data items are passed as dictionaries, the value-key is the key of the dictionary to access the items chart value.
- label-key (none,int,string): Same as the value-key but for getting an items label content.
- level-style (function,array,gradient): Level style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each level the style at the levels index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the the levels index divided by the number of levels as position on the gradient.

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

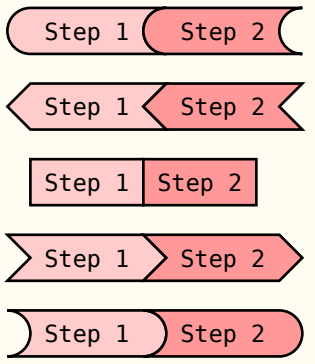
### 4.5.3 Parameters

```
pyramid(
  data,
  value-key,
  label-key,
  level-style,
  name,
  ..style
)
```

# 5 SmartArt

## 5.1 CHEVRON-CAPS

Possible chevron caps: `("(", "<", "|", ">", ")")`

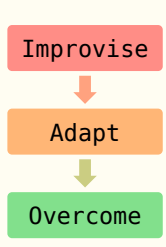



```
for cap in smartart.process.CHEVRON-CAPS {
  smartart.process.chevron(
    ([Step 1], [Step 2]), spacing: 0,
    start-cap: cap, middle-cap: cap, end-cap: cap)
  translate(y: -1)
}
```

## 5.2 Process

## 5.3 basic

Draw a basic process chart, describing sequencial steps




```
let steps = ([Improvise], [Adapt], [Overcome])
let colors = (red, orange, green).map(c => c.lighten(40%))

smartart.process.basic(
  steps,
  step-style: colors,
  equal-width: true,
  steps: (max-width: 8em),
  dir: ttb
)
```

### 5.3.1 Styling

**Root** `process-basic`

**spacing:** `number` or `length` Default: `0.2em`

Gap between steps and arrows.

**steps.radius:** `number` or `length` Default: `0.2em`

Corner radius of the steps boxes.

**steps.padding:** `number` or `length` Default: `0.6em`

Inner padding of the steps boxes.

**steps.max-width:** `number` or `length` Default: `5em`

Maximum width of the steps boxes.

**steps.shape:** `str` or `none` Default: `"rect"`

Shape of the steps boxes. One of `"rect"`, `"circle"` or `none`

**steps.fill:** `color` or `gradient` or `pattern` or `none` Default: `none`

Fill color of the steps boxes.

**steps.stroke:** `stroke` or `none` Default: `none`

Stroke color of the steps boxes.

**arrows.width:** `number` or `length` Default: `1.2em`

Width / length of arrows.

**arrows.height:** `number` or `length` Default: `1em`

Height of arrows.

**arrows.double:** `boolean` Default: `false`

Whether arrows are uni- or bi-directional.

**arrows.fill:** `string` or `color` or `gradient` or `pattern` or `none` Default: `"steps"`

Fill color of the arrows. If set to "steps", the arrows will be filled with a color in between those of the neighboring steps.

**arrows.stroke:** `stroke` or `none` Default: `none`

Stroke used for the arrows.

- steps (array): Array of steps (`<content>` or `<str>`)
- arrow-style (function, array, gradient): Arrow style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each arrow the style at the arrows index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the arrows index divided by the number of steps as position on the gradient.

  If one of stroke or fill is not in the style dictionary, it is taken from the smartarts style.
- step-style (function, array, gradient): Step style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each step the style at the steps index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the steps index divided by the number of steps as position on the gradient.

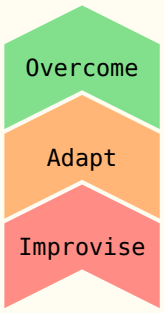
  If one of stroke or fill is not in the style dictionary, it is taken from the smartarts style.
- equal-width (boolean): If true, all steps will be sized to have the same width
- equal-height (boolean): If true, all steps will be sized to have the same height
- dir (direction): Direction in which the steps are laid out. The first step is always placed at (0, 0)

### 5.3.2 Parameters

```
basic(
  steps,
  arrow-style,
  step-style,
  equal-width,
  equal-height,
  dir,
  name,
  ..style
)
```

## 5.4 chevron

Draw a chevron process chart, describing sequencial steps




```
let steps = ([Improvise], [Adapt], [Overcome])
let colors = (red, orange, green).map(c => c.lighten(40%))

smartart.process.chevron(
  steps,
  step-style: colors,
  equal-width: true,
  steps: (max-width: 8em, cap-ratio: 25%),
  dir: btt
)
```

### 5.4.1 Styling

**Root** `process-chevron`

**spacing:** `number` or `length` Default: `0.2em`

Gap between steps.

**start-cap:** `string` Default: `">"`

Cap at the start of the process (first step). See `CHEVRON-CAPS` for possible values.

**mid-cap:** `string` Default: `">"`

Cap between steps. See `CHEVRON-CAPS` for possible values.

**end-cap:** `string` Default: `">"`

Cap at the end of the process (last step). See `CHEVRON-CAPS` for possible values.

**start-in-cap:** `boolean` Default: `false`

If true, the content of the first step is shifted inside the start cap (useful with “(“ or “<”).

**end-in-cap:** `boolean` Default: `false`

If true, the content of the last step is shifted inside the end cap (useful with “)” or “>”).

**steps.padding:** `number` or `length` Default: `0.6em`

Inner padding of the steps boxes.

**steps.max-width:** `number` or `length` Default: `5em`

Maximum width of the steps boxes.

**steps.cap-ratio:** `ratio` Default: `50%`

Ratio of the caps width relative to the steps heights (or the opposite if laid out vertically).

**steps.fill:** `color` or `gradient` or `pattern` or `none` Default: `none`

Fill color of the steps boxes.

**steps.stroke:** `stroke` or `none` Default: `none`

Stroke color of the steps boxes.

- steps (array): Array of steps (`<content>` or `<str>`)
- step-style (function, array, gradient): Step style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each step the style at the steps index modulo the arrays length gets used.

- gradient: A gradient that gets sampled for each data item using the steps index divided by the number of steps as position on the gradient.

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

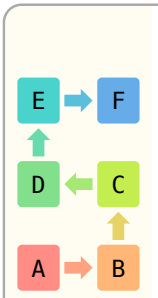
- equal-length (boolean): If true, all steps will be sized to have the same length (in the layout's direction, the other dimensions always being equal)
- dir (direction): Direction in which the steps are laid out. The first step is always placed at (0, 0)

### 5.4.2 Parameters

```
chevron(
  steps,
  step-style,
  equal-length,
  dir,
  name,
  ..style
)
```

## 5.5 bending

Draw a bending process chart, describing sequencial steps in a zigzag layout




```
let steps = ([A], [B], [C], [D], [E], [F])
let colors = (
  red, orange, yellow.mix(green), green, green.mix(blue), blue
).map(c => c.lighten(40%))

smartart.process.bending(
  steps,
  step-style: colors,
  equal-width: true,
  layout: (
    flow: (ltr, btt), max-stride: 2
  )
)
```

### 5.5.1 Styling

**Root** `process-bending`

**spacing:** `number` or `length` Default: `0.2em`

Gap between steps and arrows.

**steps.radius:** `number` or `length` Default: `0.2em`

Corner radius of the steps boxes.

**steps.padding:** `number` or `length` Default: `0.6em`

Inner padding of the steps boxes.

**steps.max-width:** `number` or `length` Default: `5em`

Maximum width of the steps boxes.

**steps.shape:** `str` or `none` Default: `"rect"`

Shape of the steps boxes. One of `"rect"`, `"circle"` or `none`

**steps.fill:** `color` or `gradient` or `pattern` or `none` Default: `none`

Fill color of the steps boxes.

**steps.stroke:** `stroke` or `none` Default: `none`

Stroke color of the steps boxes.

**arrows.width:** `number` or `length` Default: `1.2em`

Width / length of arrows.

**arrows.height:** `number` or `length` Default: `1em`

Height of arrows.

**arrows.double:** `boolean` Default: `false`

Whether arrows are uni- or bi-directional.

**layout.max-stride:** `number` Default: `3`

Maximum number of steps before turning, i.e. making a zigzag.

**layout.flow:** `array` Default: `(ltr, ttb)`

Pair of directions on different axes indicating the primary and secondary layout directions.

**arrows.fill:** `string` or `color` or `gradient` or `pattern` or `none` Default: `"steps"`

Fill color of the arrows. If set to “steps”, the arrows will be filled with a color in between those of the neighboring steps.

**arrows.stroke:** `stroke` or `none` Default: `none`

Stroke used for the arrows.

- steps (array): Array of steps (`<content>` or `<str>`)
- arrow-style (function, array, gradient): Arrow style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each arrow the style at the arrows index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the arrows index divided by the number of steps as position on the gradient.

  If one of stroke or fill is not in the style dictionary, it is taken from the smartarts style.
- step-style (function, array, gradient): Step style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each step the style at the steps index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the steps index divided by the number of steps as position on the gradient.

  If one of stroke or fill is not in the style dictionary, it is taken from the smartarts style.
- equal-width (boolean): If true, all steps will be sized to have the same width
- equal-height (boolean): If true, all steps will be sized to have the same height

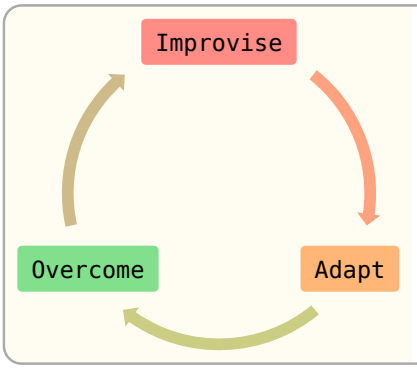
#### 5.5.2 Parameters

```
bending(
  steps,
  arrow-style,
  step-style,
  equal-width,
  equal-height,
  name,
  ..style
)
```

## 5.6 Cycle

## 5.7 basic

Draw a basic cycle chart, describing cyclic steps




```
let steps = ([Improvise], [Adapt], [Overcome])
let colors = (red, orange, green).map(c => c.lighten(40%))

smartart.cycle.basic(
  steps,
  step-style: colors,
  steps: (max-width: 5cm)
)
```

#### 5.7.1 Styling

**Root** `cycle-basic`

**steps.radius:** `number` or `length` Default: `0.2em`

Corner radius of the steps boxes.

**steps.padding:** `number` or `length` Default: `0.6em`

Inner padding of the steps boxes.

**steps.max-width:** `number` or `length` Default: `5em`

Maximum width of the steps boxes.

**steps.shape:** `str` or `none` Default: `"rect"`

Shape of the steps boxes. One of `"rect"`, `"circle"` or `none`

**steps.fill:** `color` or `gradient` or `pattern` or `none` Default: `none`

Fill color of the steps boxes.

**steps.stroke:** `stroke` or `none` Default: `none`

Stroke color of the steps boxes.

**arrows.thickness:** `number` or `length` Default: `1em`

Thickness of arrows.

**arrows.double:** `boolean` Default: `false`

Whether arrows are uni- or bi-directional.

**arrows.curved:** `boolean` Default: `false`

Whether arrows are curved or straight.

**arrows.fill:** `string` or `color` or `gradient` or `pattern` or `none` Default: `"steps"`

Fill color of the arrows. If set to "steps", the arrows will be filled with a color in between those of the neighboring steps.

**arrows.stroke:** `stroke` or `none` Default: `none`

Stroke used for the arrows.

- steps (array): Array of steps (`<content>` or `<str>`)
- arrow-style (function, array, gradient): Arrow style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each arrow the style at the arrows index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the arrows index divided by the number of steps as position on the gradient.

  If one of stroke or fill is not in the style dictionary, it is taken from the smartarts style.
- step-style (function, array, gradient): Step style of the following types:
  - function: A function of the form `index => style` that must return a style dictionary. This can be a `palette` function.
  - array: An array of style dictionaries or fill colors of at least one item. For each step the style at the steps index modulo the arrays length gets used.
  - gradient: A gradient that gets sampled for each data item using the steps index divided by the number of steps as position on the gradient.

  If one of stroke or fill is not in the style dictionary, it is taken from the smartarts style.
- equal-width (boolean): If true, all steps will be sized to have the same width
- equal-height (boolean): If true, all steps will be sized to have the same height
- ccw (boolean): If true, steps are laid out counter-clockwise. If false, they're placed clockwise. The center of the cycle is always placed at (0, 0)
- radius (number, length): The radius of the cycle
- offset-angle (angle): Offset of the starting angle

### 5.7.2 Parameters

```
basic(
  steps,
  arrow-style,
  step-style,
  equal-width,
  equal-height,
  ccw,
  radius,
  offset-angle,
  name,
  ..style
)
```