

Meander

User guide

Abstract

Meander implements a content layout algorithm to provide text threading (when text from one box spills into a different box if it overflows), uneven columns, and image wrap-around.

Feature requests

For as long as the feature doesn't exist natively in Typst (see issue: [github:typst/typst #5181](#)), feel free to submit test cases of layouts you would like to see supported by opening a new [issue](#).

Versions

- [dev](#)
- [0.2.2 \(latest\)](#)
- [0.2.1](#)
- [0.2.0](#)
- [0.1.0](#)

What's new since 0.2.1? [local font size control](#), [hyphenation and language](#), [post-threading style](#), [pagebreak and colbreak](#), [placement options](#), [overflow handlers](#)

Contents

I	Quick start	2
II	Showcase	4
III	Understanding the algorithm	5
IV	Advanced techniques	7
V	Interfacing with other content	16
VI	Modularity (WIP)	20
VII	Module details	21

I Quick start

The main function provided is `#meander.reflow`, which takes as input a sequence of “containers”, “obstacles”, and “flowing content”, created respectively by the functions `#container`, `#placed`, and `#content`. Obstacles are placed on the page with a fixed layout. After excluding the zones occupied by obstacles, the containers are segmented into boxes then filled by the flowing content.

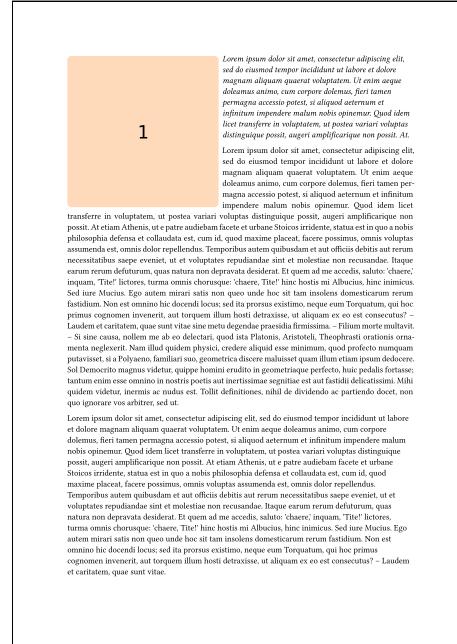
I.a A simple example

Below is a single page whose layout is fully determined by Meander. Multi-page setups are also possible, see Section IV.c.

```
#meander.reflow({
  import meander: *
  // Obstacle in the top left
  placed(top + left, my-img-1)

  // Full-page container
  container()

  // Flowing content
  content[
    _#lorem(60)_
    #[[
      #set par(justify: true)
      #lorem(300)
    ]]
    #lorem(200)
  ]
})
```



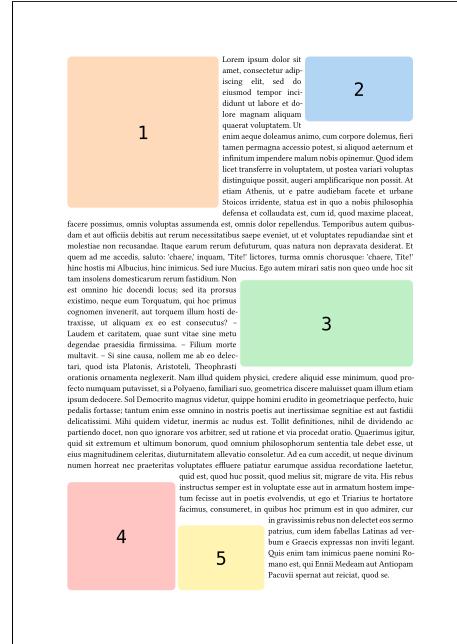
Meander is expected to respect the majority of styling options, including headings, paragraph justification, font size, etc. Notable exceptions are detailed in Section IV.b. If you find a discrepancy make sure to file it as a [bug report](#) if it is not already part of the [known limitations](#).

Note: paragraph breaks may behave incorrectly. You can insert vertical spaces if needed.

I.b Multiple obstacles

`#meander.reflow` can handle as many obstacles as you provide (at the cost of potentially performance issues if there are too many, but experiments have shown that up to ~100 obstacles is no problem).

```
#meander.reflow({  
  import meander: *  
  // As many obstacles as you want  
  placed(top + left, my-img-1)  
  placed(top + right, my-img-2)  
  placed(horizon + right, my-img-3)  
  placed(bottom + left, my-img-4)  
  placed(bottom + left, dx: 32%,  
         my-img-5)  
  
  // The container wraps around all  
  container()  
  content[  
    #set par(justify: true)  
    # lorem(430)  
  ]  
})
```



I.c Columns

In order to simulate a multi-column layout, you can provide several `container` invocations. They will be filled in the order provided.

```
#meander.reflow({
  import meander: *
  placed(bottom + right, my-img-1)
  placed(center + horizon, my-img-2)
  placed(top + right, my-img-3)

  // With two containers we can
  // emulate two columns.
  container(width: 55%)
  container(align: right, width: 40%)
  // Alternatively, a simple
  // container()
  // would also work because the first
  // container counts as occupied and
  // the second one would take all the
  // remaining space.
  content[#lorem(470)]
})
```

1

2

3

Latinus puer dolor ei moerit, conseruac
adspicere illa acta et vixim tempore recaudant
at labore et dolore magnum aliquam pauper
voluptatem. Ut prius aequo dolorem animo,
cum corpore dolens, fleri tunus permagis
accessio potest, ut aliud aeternum et infinitum
impedit, quod non posse opinatur. Quid ide
lief transuersa in voluptate, ut in natura
voluptas distinguenda possit, augeri amplifi
carique non possit. At etiam Athenis, ut e patre
audiebam facete et vixim Socratis irridente, status et
in ipsa vita amplius voluptate defensus erat, et
cum id, quod maxime placuit, facere possumus, omnis
voluptas assumenda est, omnis dolor repellendus.
Temporibus autem quibusdam et aut officiis debitis at
renuntiatur, ut si quis saepius emuletur, ut et voluptates
repudiandae sint et non possint tollandas. Inquit
euxin serum defensum, quae natura non depravata
desiderat. Et quem ad me accedit, salutis: 'clausis'
inquit. 'Tutis' hinc, turma omnis chosape:
chare, ut hinc hunc mihi Albucius, hic inimicus.
Sed hinc Moretus. Igo notandum
mirari satis non posso unde hoc
sit tam insolens domesinarium
terram fuisse, quae etiam in
hic docebat leonis se non posse
existimare, neque cum Torquatum,
qui hoc primus cognovit, inveniatur, aut torquatum illum hunc deriduisse, et aliquantum
ex excessu inveniatur. 'Dilectum et carissimum, quae sunt
ut in sine metu deponit, praecepsa firmissima.
Filium morte mulierit. — Si sine causa, nollem me ab eo
delectari, quod ita Platonicus, Aristoteli, Theophrasti
ornationis ornamenta neglexerit. Nam illud quidem
physici, et aliq. etiam in medicina, et in philosophia, et in tactico
manusq. patuerunt, si a Polymeno, familiari sue,
geometrica discere maluerit quam illum etiam ipsam
dedisse. Sed Democritus magnus videtur, quippe
homini crudito in geometria perficiuntur pedatis
fortasse, ut non esse omnino in nobis vestigia aut
invertisimae signitate est aut fastidii deliciationis. Mih
videtur, inermis et nudus est. Tolle
definitiones, nihil de dividendo ac partiendo docet, non
quo ignorat, sed quod non potest, et non potest
scire. Quiescere igitur, quid est extrellum et ultimum
bonorum, quod omnium philosophorum sententia tale
debet esse, ut eius magnitudinem celestis,
diuturnitatem alevato consuletur. Ad es cum accedit,
ut neque divinitum nomen horret nec praeferitas

voluptates effluere patitur eunomus
assidua reveratione hætitur, quid est,
quod hue possit, quod melius sit,
migratio de vita. His enim inservit
etiam etiam, ut non possit, et in
armatum hostem impetus fascis set in
poets evolvendi, ut ego et Triarius te
luctatore facimus, consumunt, in
duabus primis, ut in quo adiutor,
an in proprio rebatur, id est
eos seruo patruis, cum idem fabellas
Latinas ad verbum e Graecis expressas
non inviri legant. Quis enim tam invi
cupit, ut non possit, et non potest, et
qui Ennius Medeum aut Antropam
Pacuvi spernit aut recitat, quod
se idem Euripidis fabulis
decorat, et non possit, et non potest
oderit Synephobus, ego, inquit,
potius Cæcilius aut Andrianus Ter
enti quam stranege Menandri legam?
A quibus tantum dissentit, ut, cum
Sophocles vel optime scriptum
Electram, tamen male conversum Attilii
multi legendum patet, de quo

II Showcase

A selection of nontrivial examples of what is feasible. You can find the source code on the [repository](#), and the following sections will give you more details on how to achieve these kinds of layouts.

seminar.sty is a LaTeX style for typesetting slides or transparencies, and accompanying notes. Here are some of its special features: It is compatible with Ams-LaTeX, and you can use PostScript and Ams fonts. Slides can be landscape and portrait. There is support for color and frames. The magnification can be changed easily. Overlays can be produced from a single slide environment. Accompanying notes, such as the text of a presentation, can be put outside the slide environments. The slides, notes or both together can then be typeset in a variety of formats.

examples/5181-a/main.typ

Motivated by [github:typst/typst #5181](#) (a)



examples/5181-b/main.typ

Motivated by [github:typst/typst #5181](#) (b)

Talmudifier Test Page

Aenean sed dolor suscipit, dignissimus ut sit blandit enim. Pellentesque scelerisque viverra ligula, vel congue lorem sem non. Nulla nec convallis neque. Donec ut leo velit. Donec fames odio, tincidunt non vestibulum ut, consectetur eget lectus. Donec varius finibus scelerisque aliquam. Igneodolus odio sollicitudin eget sem non, tincidunt ut aliquam sollicitudin scelerisque. Nam et arcu dictum ultricies. Proin vel ante ipsum, arcu etiam sollicitudin occidit. Nul- lam ultricies tempor neque, varius scelerisque neque. Nam et arcu eu odio maximus tempor et sit amet elit. Et Morbi fermentum dapibus elementum. Proin id metus ipsum. Aenean posuere nunc quis lacus varius, eget molestie mauris accumsan. *[I] Fasces sapien ipsum, cursus a tincidunt vel, dignissim ut mi.* Morbi id velit ac turpis ullamcorper lacunae. *[Cras bibendum id lobortis vivat eros tempus scelerisque. Vivamus sed pellentesque elit, non imperdiet massa. Curabitur dicitur etiam auctor, non facilisis.* Vestibulum id pellentesque id, sit amet etiam facilisis. Elitis a nunc dolor. *[R] In hac vel tempus, libero maenae et, blandit massa. Aliquam quisque id neque et, sit amet ultrices. Integer id justo est. Quisque lectus erat etiam aliquam faucibus. Etiam ut orci leo. Nulla dignissim erat pubian et facilisis, ac venenatis leo tincidunt. Quisque ut lorem tortor. Quisque nec porttitor elit. Ut finibus ullamcorper odio, in porttitor lorem suscipit ut, maenae blandit, efficiendi odio velit, dictum cuncto, ac maenae efficiendi libritus libro, eget commmodo purum, consequetum elementum magna.¹ *[tempus⇱6⇱0⇱]* Sed purus lec-tus, > Check out how nice that little etiam venenatis nisi nec tempus fringilla. Vivamus intere. Integer ac urna et lectus ut tempus mattis ac id porttitor que. Donec ut ante, semper non nunc finibus sit amet felis ut, auptate eiusmod nunc.*

examples/talmudifier/main.typ

From [github:subalterngames/talmudifier](https://github.com/subalterngames/talmudifier)
Motivated by [github:typst/typst #5181 \(c\)](https://github.com/typst/typst/pull/5181)

III Understanding the algorithm

The same page setup as the previous example I.c will internally be separated into

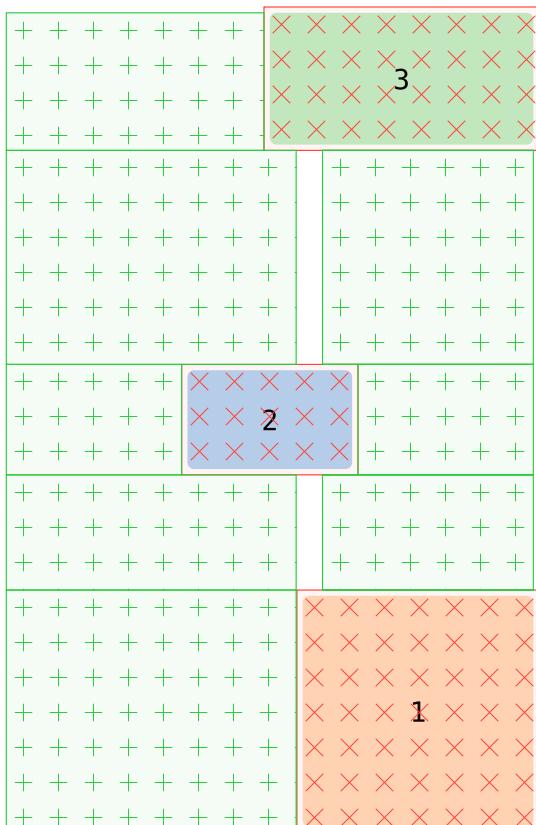
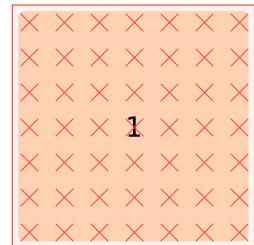
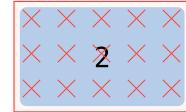
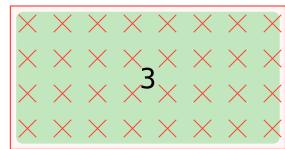
- obstacles `my-img-1`, `my-img-2`, and `my-img-3`.
- containers `#(x: 0%, y: 0%, width: 55%, height: 100%)` and `#(x: 60%, y: 0%, width: 40%, height: 100%)`
- flowing content `#lorem(470)`.

Initially obstacles are placed on the page (→). If they have a boundary parameter, it recomputes the exclusion zone.

Then the containers are placed on the page and segmented into rectangles to avoid the exclusion zones (↓).

Finally the flowing content is threaded through those boxes (↘), which may be resized vertically a bit compared to the initial segmentation.

The debug views on this page are accessible via `#meander.regions` and `#meander.reflow.with(debug: true)`



...Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit animus.

...carique non possit. At etiam Athenis, ut e patre audiebam facete et urbanos Stoicos irridente, statua est in quo a nobis philosophia defensa et officia debitis in renum necessitatibus saepe eveniet, ut et voluptates repudiandas sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedit, saluto: 'chaere', inquam, 'Tite!' licet, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucus, hinc inimicus

...sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen in-

...venierit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? — Laudem et caritatem, quae sunt virtus sine metu degendae praesidia firmissima. —

...Filium morte multavit. — Si sine causa, nolle ab eo delectari, quod ista Platonis, Aristotelis, Theophrasti orationis ornamenta neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto-

...nunquam putavisset, si a Polyaeno, familiari suo, geometrica discere malisset quam illum etiam ipsum dedocere. Sol Democritus magnus videtur, quippe homini eruditio in geometria perfecto, huic pedalis fortasse; tantum enim esse omnino in nostris poëtis aut in rhetoribus segnitiae est aut fastidii delicatissimi. Mihi quidem videtur, inermis ac nudus est. Tollit

definitions, nihil de dividendo a partiendo docet, non

quo ignore vos arbitreris, sed ut ratione et via procedat

præceptio. Quærimus igitur, quid sit extreum et ultimum

bonorum, quod omnium philosophorum sententia tale

debet esse, ut eius magnitudinem celestis,

diuturnitatem allevatio consoletur. Ad ea cum accedit,

ut neque divinum numen horreat nec praeteritas

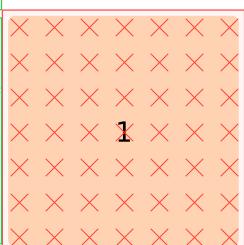
cus paene nomini Romano est, qui Enni Medeam aut Antioepam

Pacuvii spernat aut reicit, quod se idem Euripidis fabulis

delectari dicat, Latinas litteras oderit? Synephebos ego, inquit, patius Caecilius aut Andriam Ter-

entii quam utramque Menandri legam?

A quibus tantum dissentio, ut, cum Sophocles vel optime scripsit Electram, tamen male conversam Atilii mihil legendam putem, de quo.



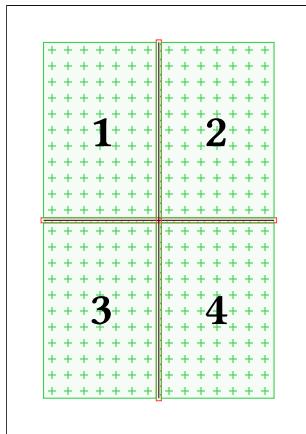
The order in which the boxes are filled is in the priority of

- container order
- top → bottom
- left → right

which has implications for how your text will be laid out. Indeed compare the following situations that result in the same boxes but in different orders:

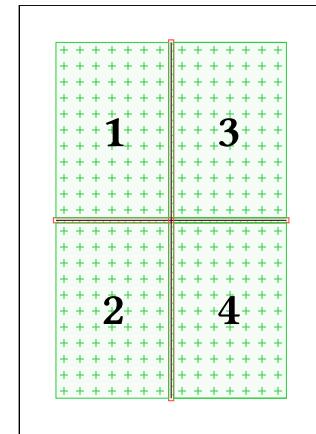
```
#meander.regions({
  import meander: *
  placed(center + horizon,
    line(end: (100%, 0%)))
  placed(center + horizon,
    line(end: (0%, 100%)))

  container(width: 100%)
})
```



```
#meander.regions({
  import meander: *
  placed(center + horizon,
    line(end: (100%, 0%)))
  placed(center + horizon,
    line(end: (0%, 100%)))

  container(width: 50%)
  container(align: right, width: 50%)
})
```



And even in the example above, the box **1** will be filled before the first line of **2** is used. In short, Meander **does not “guess” columns**. If you want columns rather than a top-bottom and left-right layout, you need to specify them.

IV Advanced techniques

IV.a Obstacle contouring

Although Meander started as only a text threading engine, the ability to place text in boxes of unequal width has direct applications in more advanced paragraph shapes. This has been a desired feature since at least [issue #5181](#).

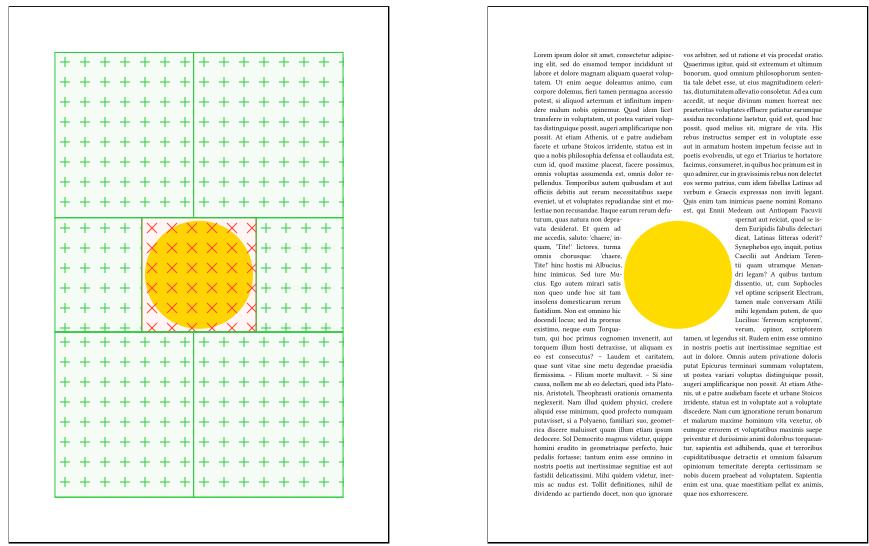
Even though this is somewhat outside of the original feature roadmap, Meander makes an effort for this application to be more user-friendly, by providing functions to redraw the boundaries of an obstacle. Here we walk through these steps.

Basic contouring is simply the ability to customize the margin around obstacles by passing to `#placed` the argument boundary: `contour.margin(1cm)`. `#contour.margin` also accepts parameters `x`, `y`, `top`, `bottom`, `left`, `right`, with the precedence you would expect, to customize the margins precisely.

As equations

Here is our starting point: a simple double-column page with a cutout in the middle for an image.

```
#meander.reflow({
  import meander: *
  placed(center + horizon) [
    #circle(
      radius: 3cm,
      fill: yellow,
    )
  ]
  container(
    width: 50% - 3mm,
    margin: 6mm,
  )
  container()
  content[
    #set par(justify: true)
    #lorem(590)
  ]
})
```



Meander sees all obstacles as rectangular, so the circle leaves a big ugly [square hole](#) in our page.

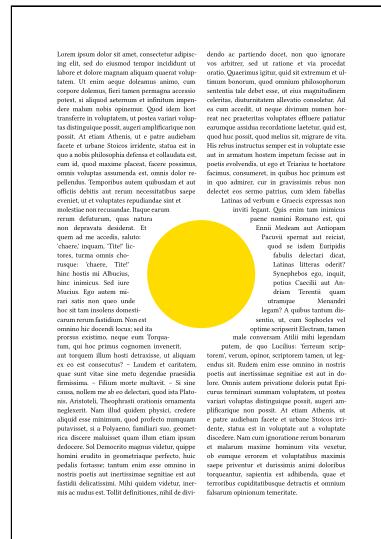
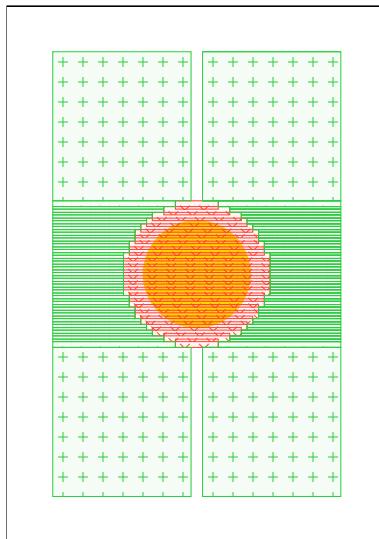
Fear not! We can redraw the boundaries. `#meander.placed` accepts as parameter boundary a sequence of box transformers to change the way the object affects the layout. These transformations are normalized to the interval [0, 1] for convenience. The default boundary value is `#contour.margin(5pt)`.

`#meander.contour.grid` is one such redrawing function, from $[0, 1] \times [0, 1]$ to `bool`, returning for each normalized coordinate (x, y) whether it belongs to the obstacle.

So instead of placing directly the circle, we write:

```
#meander.reflow({
    import meander: *
    placed(
        center + horizon,
        boundary:
            // Override the default margin
            contour.margin(1cm) +
            // Then redraw the shape as a grid
            contour.grid(
                // 25 vertical and horizontal subdivisions.
                // Just pick a number that looks good.
                // A good rule of thumb is to start with obstacles
                // about as high as one line of text.
                div: 25,
                // Equation for a circle of center (0.5, 0.5) and radius 0.5
                (x, y) => calc.pow(2 * x - 1, 2) + calc.pow(2 * y - 1, 2) <= 1
            ),
            // Underlying object
            circle(radius: 3cm, fill: yellow),
    )
    // ...
})
```

This results in the new subdivisions of containers below.



This enables in theory drawing arbitrary paragraph shapes. Note the high density of obstacles on the debug view above: this works here but we are getting close to the resolution limit of meander, so don't try to draw obstacles with a resolution too much lower than the normal line height.

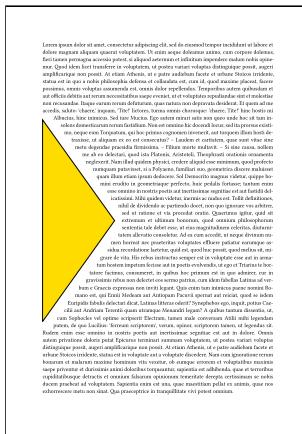
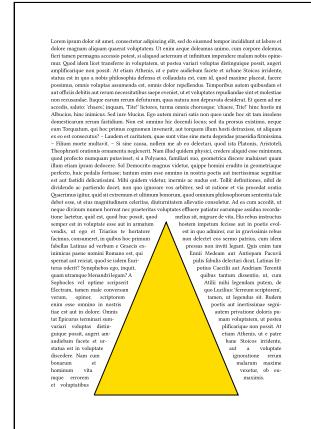
As stacked rectangles

If your shape is not convenient to express through a grid function, but has some horizontal or vertical regularity, here are some other suggestions:



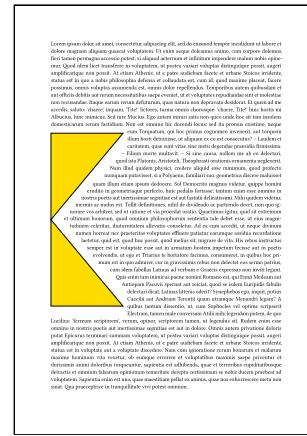
```
#meander.reflow({
  import meander: *
  placed(right + bottom,
  boundary:
    // The right aligned edge makes
    // this easy to specify using
    // `horiz`
    contour.horiz(
      div: 20,
      // (left, right)
      y => (1 - y, 1),
    ) +
    // Add a post-segmentation margin
    contour.margin(5mm)
  ) [...]
  // ...
})
```

```
#meander.reflow({
  import meander: *
  placed(center + bottom,
  boundary:
    // This time the vertical symmetry
    // makes `width` a good match.
    contour.width(
      div: 20,
      flush: center,
      // Centered in 0.5, of width y
      y => (0.5, y),
    ) +
    contour.margin(5mm)
  ) [...]
  // ...
})
```



```
#meander.reflow({
  import meander: *
  placed(left + horizon,
  boundary:
    contour.height(
      div: 20,
      flush: horizon,
      x => (0.5, 1 - x),
    ) +
    contour.margin(5mm)
  ) [...]
  // ...
})
```

```
#meander.reflow({
  import meander: *
  placed(left + horizon,
    boundary:
      contour.horiz(
        div: 25,
        y => if y <= 0.5 {
          (0, 2 * (0.5 - y))
        } else {
          (0, 2 * (y - 0.5))
        },
      ) +
      contour.margin(5mm)
  )[...]
  // ...
})
```



To summarize,

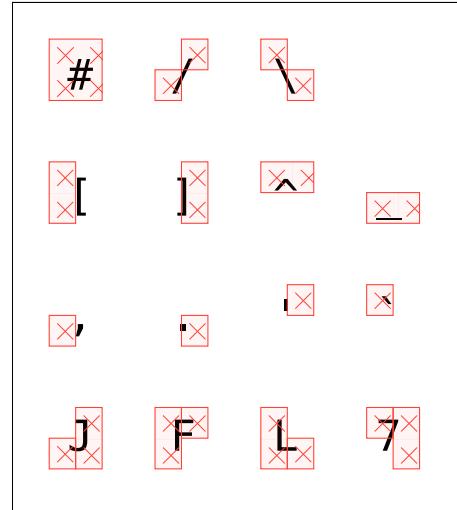
- `vert(div: _, fun)`: subdivide vertically in `div` sections, then `fun(x) = (top, bottom)` produces an obstacle between top and bottom.
- `height(div: _, flush: _, fun)`: subdivide vertically in `div` sections, then `fun(x) = (anchor, height)` produces an obstacle of height `height`, with the interpretation of `anchor` depending on the value of `flush`:
 - if `flush = top` then `anchor` will be the top of the obstacle;
 - if `flush = bottom` then `anchor` will be the bottom of the obstacle;
 - if `flush = horizon` then `anchor` will be the center of the obstacle.
- `horiz`: a horizontal version of `vert`.
- `width`: a horizontal version of `height`.

All of these functions operate on values normalized to $[0, 1]$.

As ASCII art

Another method of specifying contours is by drawing a rough shape of the obstacle in ASCII art. Pass a code block made of the characters `# / \ [] ^ _ . ' ` J F L 7` to the contouring function `#contour.ascii-art`, and you can easily draw the shape of your image.

```
#meander.reflow(debug: true, {
  import meander: *
  placed(top + left,
    boundary: contour.margin(6mm) +
    contour.ascii-art(
      `#
      # / \
      [ ] ^ _
      , . ' `
      J F L 7
    )
  )[#image]
})
```



See [examples/5181-b/main.typ](#) for a nontrivial use-case.

Remarks

The contouring functions available should already cover a reasonable range of use-cases, but if you have other ideas you could always try to submit one as a new [issue](#).

There are of course limits to this technique, and in particular increasing the number of obstacles will in turn increase the number of boxes that the layout is segmented into. This means

- performance issues if you get too wild (though notice that having 20+ obstacles in the previous examples went completely fine, and I have test cases with up to ~100)
- text may not fit in the boxes, and the vertical stretching of boxes still needs improvements.

In the meantime it is discouraged to use a subdivision that results in obstacles much smaller than the font height.

IV.b Styling

Meander respects most styling options through a dedicated content segmentation algorithm. Bold, italic, underlined, stroked, highlighted, colored, etc. text is preserved through threading, and easily so because those styling options do not affect layout much.

There are however styling parameters that have a consequence on layout, and some of them require special handling. Some of these restrictions may be relaxed or entirely lifted by future updates.

Paragraph justification

In order to properly justify text across boxes, Meander needs to have contextual access to `#par.justify`, which is only updated via a `#set` rule.

As such **do not** use `#par(justify: true)[...]`.

Instead prefer `#[#set par(justify: true); ...]`, or put the `#set` rule outside of the invocation of `#meander.reflow` altogether.

Wrong

Correct

Correct

```
#meander.reflow({
  // ...
  content[
    #par(justify: true)[
      #lorem(600)
    ]
  ]
})
```

```
#meander.reflow({
  // ...
  content[
    #set par(justify: true)
    #lorem(600)
  ]
})
```

```
#set par(justify: true)
#meander.reflow({
  // ...
  content[
    #lorem(600)
  ]
})
```



Font size and paragraph leading

The font size indirectly affects layout because it determines the spacing between lines. When a linebreak occurs between containers, Meander needs to manually insert the appropriate spacing there. Since the spacing is affected by font size, make sure to update the font size outside of the `#reflow` invocation if you want the correct line spacing. Alternatively, `size` can be passed as a parameter of `content` and it will be interpreted as the text size.

Analogously, if you wish to change the spacing between lines, use either a `#set par(leading: 1em)` outside of `#reflow`, or pass `leading: 1em` as a parameter to `content`.

Wrong

```
#meander.reflow({  
  // ...  
  content[  
    #set text(size: 30pt)  
    #lorem(80)  
  ]  
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua querat volupatem. Ut enim aequae dolore, fieri animo, cum corpore dolumus, fieri tamen penaqua accessit potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod item fieri transire in volupatem, ut postea variari voluptas distinguere possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia de- fensa et.

Correct

```
#set text(size: 30pt)
#meander.reflow({
    // ...
    content[
        #lorem(80)
    ]
})
```

Lore ipsum dolor sit amet, consectetur adipiscit elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua quaerat volupatem. Ut enim aquae doleamus animo, cum corpore dolemus, fieri tamen permangia accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in volupatem, ut postea variari voluptas distinguere possit, augeri amplificare non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et.

Correct

```
#meander.reflow({
  // ...
  content(size: 30pt) [
    #lorem(80)
  ]
})
```

Lore ipsum dolor sit amet, con-
secetur adipiscing elit, sed do eius-
mod tempor incididunt ut labore et
dolore magna aliqua querat
voluptatem. Ut enim aeque dole-
mus animo, cum corpore dolens,
fieri tamen permagna accessio
potest, si aliquod aeternum et infinitum
impedire malum nobis opinem-
ur. Quod idem licet transferri
in voluptatem, ut postea variari volup-
tas distinguere possit, augeri ampli-
ficarique non possit. At etiam
Athenis, ut e patre audiebam facete
et urbani Stoicos irridente, statua
est in quo a nobis philosophia de-
fensa et.

Hyphenation and language

Hyphenation can only be fetched contextually, and highly influences how text is split between boxes. Language indirectly influences layout because it determines hyphenation rules. To control the hyphenation and language, use the same approach as for the text size: either `#set` them outside of `#reflow`, or pass them as parameters to `#content`.

Wrong

```
#meander.reflow({
  // ...
  content[
    #set text(hyphenate: true)
    #lorem(70)
  ]
})
```

Lore ipsum dolor sit amet, consectetur adipisciing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua querat volupatatem. Ut enim aequae doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferri in volupatatem, ut postea variari augeri distinguere possit, variari augeri amplificari non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

Correct

```
#set text(hyphenate: true)
#meander.reflow({
  // ...
  content[
    #lorem(70)
  ]
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Quaerat volupatem. Ut enim aequo doleamus animo, cum corpore dolum, fieri tamen permane accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in volupatem, ut postea variari voluptas distingue possit, augeri amplificarique non possit. At etiam Athenis, ut patre audiebam facete et urbane Stoicos.

Correct

```
#meander.reflow({
  // ...
  content(hyphenate: true)[
    #lorem(70)
  ]
})
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Quaerat voluptatem. Ut enim aequae doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinetur. Quod idem licet transference in voluptatem, ut postea variari voluntas distinguebitur, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoeicos.

Applying styles to containers

`#container` accepts a style dictionary that may contain the following keys:

- `text-fill`: the color of the text in this container,
- `align`: the left/center/right alignment of content,
- and more to come.

These options have in common that they do not affect layout so they can be applied post-threading to the entire box. Future updates may lift this restriction.

```
#meander.reflow({
  import meander: *
  container(width: 25%,
    style: (align: right, text-fill: blue))
  container(width: 75%,
    style: (align: center))
  container(
    style: (text-fill: red))
  content[#lorem(590)]
})
```



IV.c Multi-page setups

Meander can deal with text that spans multiple pages, you just need to place `#pagebreaks` appropriately. Note that `#pagebreak` only affects the obstacles and containers, while `#content` blocks ignore them entirely.

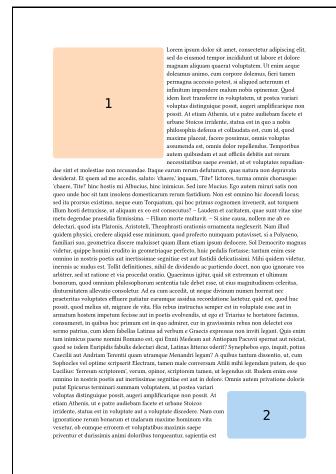
```
#meander.reflow({
  import meander: *

  placed(top + left, my-img-1)
  placed(bottom + right, my-img-2)
  container()

  pagebreak()

  placed(top + right, my-img-3)
  placed(bottom + left, my-img-4)
  container(width: 45%)
  container(alignment: right, width: 45%)

  content[#lorem(1000)]
})
```



If you run into performance issues, consider finding spots where you can break the `#reflow` invocation. As long as you don't insert a `#pagebreak` explicitly, several `#reflows` can coexist on the same page.

```

//#set block(spacing: 0em)
// First half-page
#meander.regions({
  import meander: *
  placed(top + left, my-img-1)
  container(height: 45%)
})

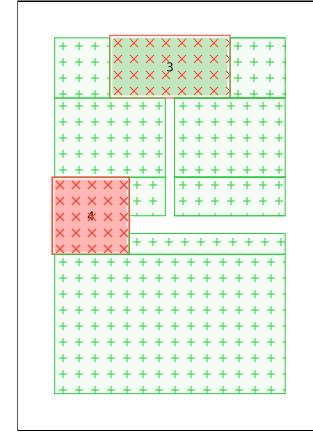
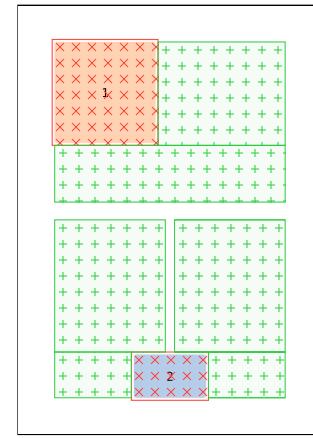
// Overflows on the second page
#meander.regions({
  import meander: *
  placed(bottom + center, my-img-2)
  container(align: bottom, height: 50%, width: 48%)
  container(align: bottom + right,
            height: 50%, width: 48%)

  pagebreak()

  placed(top + center, my-img-3)
  placed(horizon, my-img-4)
  container(height: 50%, width: 48%)
  container(align: right, height: 50%, width: 48%)
})

// Takes over for the last half-page
#meander.regions({
  import meander: *
  // This obstacle is already placed by the previous
  // invocation. We just restate it without displaying
  // it so that it appears only once yet still gets
  // counted as an obstacle for both invocations.
  placed(display: false, horizon, my-img-4)
  container(align: bottom, height: 45%)
})

```



Note: the default behavior if the content provided overflows the available containers is **not** to put in on the next page, but rather to show a warning. You can manually enable the overflow going to the next page by passing to `#reflow` the parameter `overflow: pagebreak`. Other options include `overflow: panic` if you want accidental overflows to trigger an immediate panic.

Analogously, `#colbreak` breaks to the next container. Note that `#pagebreak` is a *container* separator while `#colbreak` is a *content* separator. The next container may be on the next page, so the right way to create an entirely new page for both containers and content is a `#pagebreak` **and** a `#colbreak`.

```

#meander.reflow({
  import meander: *

  container(width: 50%, style: (text-fill: red))
  container(style: (text-fill: blue))
  content[#lorem(100)]
  colbreak()
  content[#lorem(500)]

  pagebreak()
  colbreak()

  container(style: (text-fill: green))
  container(style: (text-fill: orange))
  content[#lorem(400)]
  colbreak()
  content[#lorem(400)]
  colbreak() // Note: the colbreak applies only after the overflow is handled.

  pagebreak()

  container(align: center, dy: 25%, width: 50%, style: (text-fill: fuchsia))
  container(width: 50% - 3mm, margin: 6mm, style: (text-fill: teal))
  container(style: (text-fill: purple))
  content[#lorem(400)]
})

```

Etiam in Athenis, et a patre audiebam facete et urbane Stoicos irridente, status est in quo a nobis philosophia defensata et collaudata est, cum id, quod maxime placet, facere possumus, omnibus sollicitus erat, omnisi dolor repellendus. Temporibus autem quibusdam et.

alii fortasse: tantum enim estesse omnino in nostris poëtis aucti meritissima est aut fastidii deliciosa. Milius quidam velut, nemis ac tunc emi. Tantum enim estesse omnino, nihil devenit auctoritate, non quod est esse omnino, sed ut ratione et via procedat oratio. Quoniam igit; sicut et extremum et ultimum orationis quod omnia philosophorum sententia tale debet esse, non potest esse nisi ratione et via procedat, dimitur autem alioconvenit. Ad eum accedit, ut neque divinum nimis horret ne præteritas volupte efficerat patitur exasperare uestimenta, sed neque hinc huiusmodi. His uestimentis instructus semper est in voluptate esse aut in armis hostium impetus fuisse ac in poëtis evolvens, ut eum et Trianus te hortator facias, nonne hoc primus est qui ad admirare, cur in gravissimis rebus non detectus es sermo patius, cum
se pateras, cum
me accidit, salutis
littere, tunc omnis
te' hinc bonus
sed. Sicut Muses. Ego
nonne hoc sit tam
bonum? Nonne
li hec; sed ita prorsus
Tropaeum, qui his primis
ut tequem illum hosti
victoriam, nonne
sime vitis sine metu
missima. — Filium morte
sa, mollem ab eo
componit. Aristoteli
unumq[ue] neglegit.
nisi, credere aliquid esse

extremum et
omnino, quod
philosophorum
sententia hinc
magis utrilibet
complacuisse
allevatio constat.
Ad ea cum accedit,
ut neque divinum
hinc huiusmodi.
præteritas volu-
ptate efficerat
extremum uestimenta
non potest esse
aut et spes
sunt, migrares
deinde et inservias
semper et in hosti
hinc huiusmodi.
hostium impetus fuisse ac in poëtis
evolvens, ut eum et Trianus te hortator
facias, consumetur, in quo his lucis
in quo admires, cur in gravissimis rebus
non detectus es sermo patius, cum

V Interfacing with other content

This section explains the parameters that are available to make a `#reflow` invocation better integrate with surrounding content.

V.a Placement

Placement options control how a `#reflow` invocation is visible by and sees other content.

page

The default, and least opinionated, mode is `placement: page`.

- suitable for: one or more pages that `meander` has full control over.
- advantages: supports pagebreaks, several invocations can be superimposed, flexible.
- drawbacks: superimposed with content that follows.

box

The option `placement: box` will emit non-placed boxes to simulate the actual space taken by the `meander`-controlled layout.

- suitable for: an invocation that is part of a larger page.
- advantages: supports pagebreaks, content that follows is automatically placed after.
- drawbacks: cannot superimpose multiple invocations.

Here is a layout that is not (as easily) achievable in `page` as it is in `box`:

```
#lorem(100)

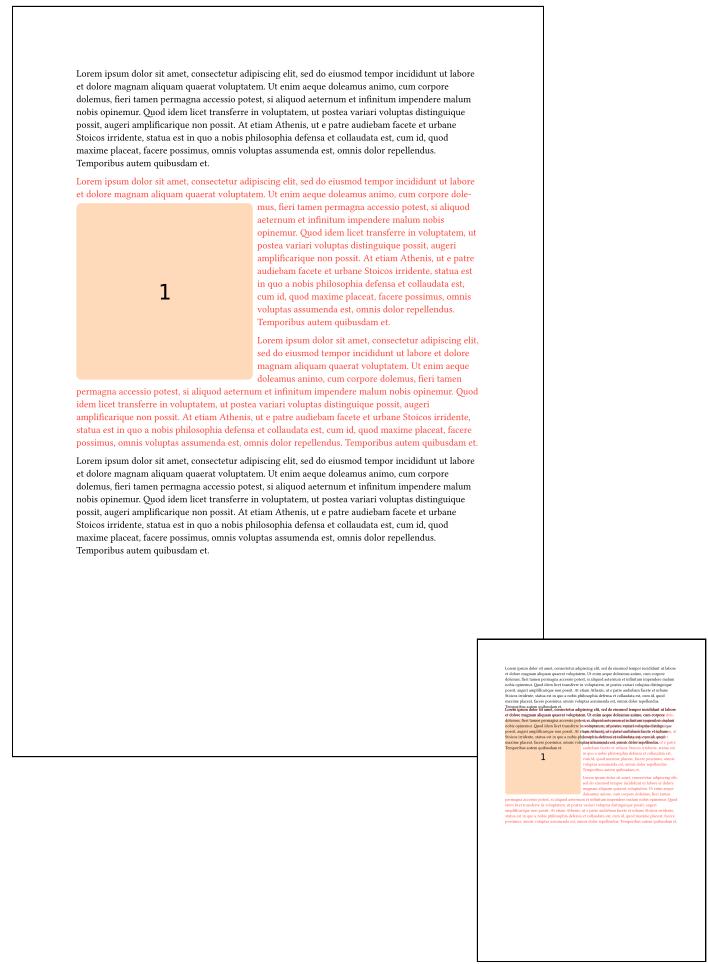
#meander.reflow(placement: box, {
    import meander: *
    placed(top + left, dy: 1cm,
        my-img-1)
    container()
    content[
        #set text(fill: red)
        #lorem(100)

        #lorem(100)
    ]
}

#lorem(100)
```

Only text in red is actually controlled by `meander`, the rest is naturally placed before and after. This makes it possible to hand over to `meander` only a few paragraphs where a complex layout is required, then fall back to the native Typst layout engine.

For reference, to the right is the same page if we omit `placement: box`, where we can see a glitchy superimposition of text.



float

Finally, `placement: float` produces a layout that spans at most a page, but in exchange it can take the whole page even if some content has already been placed.

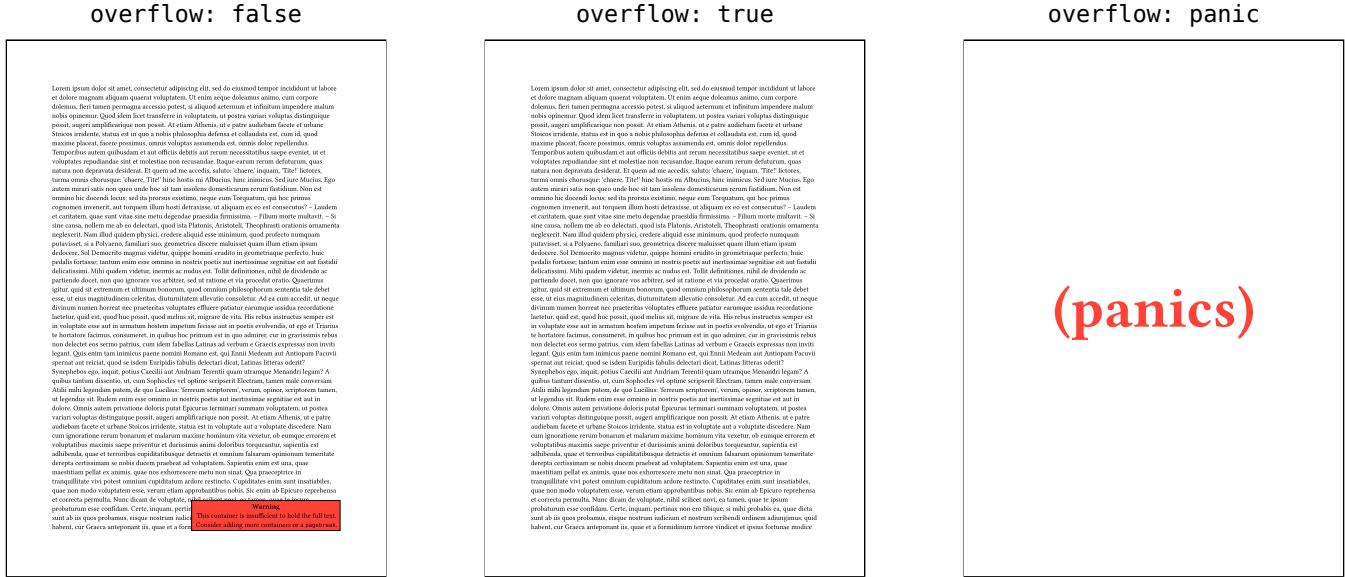
- suitable for: single page layouts.
- advantages: gets the whole page even if some content has already been written.
- drawbacks: does not support pagebreaks, does not consider other content.

V.b Overflow

When the content overflows the current container, you may specify how you want that to be handled. There are essentially 3 ways of doing that: prevent overflow, overflow to a predefined layout, or overflow to a custom layout.

No overflow

The default behavior is `overflow: false` because it avoids panics while still alerting that something is wrong. The red warning box suggests adding more containers or a pagebreak to fit the remaining text. Setting `overflow: true` will silently ignore the overflow, while `overflow: panic` will immediately abort compilation.



```
#meander.reflow(  
  overflow: false, {  
    import meander: *  
    container()  
    content[#lorem(1000)]  
  })
```

```
#meander.reflow(  
  overflow: true, {  
    import meander: *  
    container()  
    content[#lorem(1000)]  
  })
```

```
#meander.reflow(  
  overflow: panic, {  
    import meander: *  
    container()  
    content[#lorem(1000)]  
  })
```

Predefined layouts

The above options are more useful if you absolutely want the content to fit in the defined layout. A commonly desired behavior is for the overflow to simply integrate with the layout as gracefully as possible. That is the purpose of the two options that follow.

With `overflow: pagebreak`, any content that overflows is placed on the next page. This is typically most useful in conjunction with `placement: page`, and is outright incompatible with `placement: float` (see Section V.a).

```
#meander.reflow(
  overflow: pagebreak, {
    import meander: *
    container(
      width: 48%,
      style:
        (text-fill: blue),
    )
    container(
      align: right,
      width: 48%,
      style:
        (text-fill: blue),
    )
    content[#lorem(1000)]
  ) \
#text(fill: red)[
  #lorem(100)
]
```

As shown above from the fact that it does not receive the `text-fill: blue` styling, the text that overflows to the second page is not controlled by `meander`.

As for `overflow: text`, it is similarly best suited in conjunction with `placement: box`, and simply writes the text after the end of the layout.

```
#meander.reflow(placement: box, overflow: text, { import meander: * container( width: 48%, height: 50%, style: (text-fill: blue), ) container( width: 48%, height: 50%, align: right, style: (text-fill: blue)) content[#lorem(1000)] }) #text(fill: red)[ #lorem(100) ]
```

In both cases, any content that follows the `#reflow` invocation will more or less gracefully follow after the overflowing text, possibly with the need to slightly adjust paragraph breaks if needed.

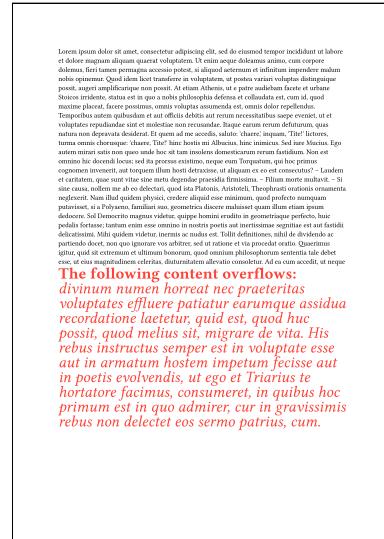
Custom layouts

If your desired output does not fit in the above predefined behaviors, you can fall back to writing a custom overflow handler. Any function that returns content can serve as handler, including another invocation of `#reflow`. This function will be given as input a dictionary with fields:

- `styled` has all styling options applied and is generally what you should use,
 - `structured` is suitable for placing in another `#reflow` invocation,
 - `raw` uses an internal representation that you can iterate over, but that is not guaranteed to be stable. Use as last resort only.

For example here is a handler that adds a header and some styling options to the text that overflows:

```
#meander.reflow(  
  placement: box,  
  overflow: tt => [  
    #set text(fill: red, size: 25pt)  
    *The following content overflows:  
    _#{tt.styled}_  
  ], {  
  import meander: *  
  container(height: 50%)  
  content[#lorem(400)]  
)
```



See also an answer I gave to [issue #1](#) which demonstrates how passing a `#reflow` layout as overflow handler can achieve layouts not otherwise supported. These applications should remain last-resort measures, and if you find that a layout you would want to achieve is only possible by chaining together several `#reflow` handlers, consider instead [reaching out](#) to see if there is a way to make this layout better supported.

VI Modularity (WIP)

Because meander is cleanly split into three algorithms (content segmentation, page segmentation, text threading), there are plans to provide

- additional configuration options for each of those steps
- the ability to replace entirely an algorithm by either a variant, or a user-provided alternative that follows the same signature.

VII Module details

VII.a Geometry (geometry.typ)

Generalist functions for 1D and 2D geometry.

- `clamp()`
- `between()`
- `intersects()`
- `resolve()`
- `align()`

clamp

Bound a value between `min` and `max`. No constraints on types as long as they support inequality testing.

Parameters

```
clamp(  
    val: any,  
    min: any none,  
    max: any none  
) -> any
```

val any

Base value.

min any or none

Lower bound.

Default: none

max any or none

Upper bound.

Default: none

between

Testing `a <= b <= c`, helps only computing `b` once.

Parameters

```
between(  
    a: length,  
    b: length,  
    c: length  
) -> bool
```

a length

Lower bound.

b length

Tested value.

c length

Upper bound. Asserted to be $\geq c$.

intersects

Tests if two intervals intersect.

Parameters

```
intersects(  
    i1: (length, length),  
    i2: (length, length),  
    tolerance: length  
)
```

i1 (length, length)

First interval as a tuple of (low, high) in absolute lengths.

i2 (length, length)

Second interval.

tolerance length

Set to nonzero to ignore small intersections.

Default: 0pt

resolve

Converts relative and contextual lengths to absolute. The return value will contain each of the arguments once converted, with arguments that contain 'x' or start with 'w' being interpreted as horizontal, and arguments that contain 'y' or start with 'h' being interpreted as vertical.

```
#context resolve(  
    width: 100pt, height: 200pt),  
    x: 10%, y: 50% + 1pt,  
    width: 50%, height: 5pt,  
)
```

(x: 10pt, y: 101pt, width: 50pt, height: 5pt)

Parameters

```
resolve(  
    size: (width: length, height: length),  
    ..args: dictionary  
) -> dictionary
```

size `(width: length, height: length)`

Size of the container as given by the `layout` function.

align

Compute the position of the upper left corner, taking into account the alignment and displacement.

Parameters

```
align(  
  alignment: alignment,  
  dx: relative,  
  dy: relative,  
  width: relative,  
  height: relative  
) -> (x: relative, y: relative)
```

alignment `alignment`

Absolute alignment.

dx `relative`

Horizontal displacement.

Default: `0pt`

dy `relative`

Vertical displacement.

Default: `0pt`

width `relative`

Object width.

Default: `0pt`

height `relative`

Object height.

Default: `0pt`

VII.b Tiling (`tiling.typ`)

Page splitting algorithm.

- `placed()`
- `container()`
- `pagebreak()`
- `colbreak()`
- `content()`
- `pat-forbidden()`
- `pat-allowed()`
- `elem-of-placed()`

- `elem-of-container()`
- `add-auto-margin()`
- `next-elem()`
- `push-elem()`
- `separate()`
- `regions()`

placed

Core function to create an obstacle.

Parameters

```
placed(
  align: alignment,
  dx: relative,
  dy: relative,
  boundary: (..function,),
  display: bool,
  content: content
) -> obstacle
```

`align` `alignment`

Reference position on the page (or in the parent container).

`dx` `relative`

Horizontal displacement.

Default: `0% + 0pt`

`dy` `relative`

Vertical displacement.

Default: `0% + 0pt`

`boundary` `(..function,)`

An array of functions to transform the bounding box of the content. By default, a 5pt margin. See `contour.typ` for a list of available functions.

Default: `(auto,)`

`display` `bool`

Whether the obstacle is shown. Useful for only showing once an obstacle that intersects several invocations.

Contrast the following:

- `boundary: contour.phantom` will display the object without using it as an obstacle,
- `display: false` will use the object as an obstacle but not display it.

Default: `true`

`content` `content`

Inner content.

container

Core function to create a container.

Parameters

```
container(  
  align: alignment,  
  dx: relative,  
  dy: relative,  
  width: relative,  
  height: relative,  
  style: dictionary,  
  margin: length  
) -> container
```

align alignment

Location on the page.

Default: top + left

dx relative

Horizontal displacement.

Default: 0% + 0pt

dy relative

Vertical displacement.

Default: 0% + 0pt

width relative

Width of the container.

Default: 100%

height relative

Height of the container.

Default: 100%

style dictionary

Styling options for the content that ends up inside this container. If you don't find the option you want here, check if it might be in the `style` parameter of `content` instead.

- `align`: flush text left/center/right
- `text-fill`: color of text

Default: (:)

margin length

Margin around the eventually filled container so that text from other paragraphs doesn't come too close.

Default: 5mm

pagebreak

Continue layout to next page.

Parameters

`pagebreak()`

colbreak

Continue content to next container.

Parameters

`colbreak()`

content

Core function to add flowing content.

Parameters

```
content(  
  data: content,  
  size: length,  
  lang: string,  
  hyphenate: bool,  
  leading: length  
) -> flowing
```

data `content`

Inner content.

size `length`

Applies `#set text(size: ...)`.

Default: `auto`

lang `string`

Applies `#set text(lang: ...)`.

Default: `auto`

hyphenate `bool`

Applies `#set text(hyphenate: ...)`.

Default: `auto`

leading `length`

Applies `#set par(leading: ...)`.

Default: `auto`

pat-forbidden

Pattern with red crosses to display forbidden zones.

Parameters

```
pat-forbidden(sz: length) -> pattern
```

sz `length`

Size of the tiling.

pat-allowed

Pattern with green pluses to display allowed zones.

Parameters

```
pat-allowed(sz: length) -> pattern
```

sz `length`

Size of the tiling.

elem-of-placed

See: `next-elem` to explain `data`. This function computes the effective obstacles from an input object, as well as the display and debug outputs.

Parameters

```
elem-of-placed(
  data: opaque,
  obj: placed
) -> elem
```

data `opaque`

Internal state.

obj `placed`

Object to measure, pad, and place.

elem-of-container

See: `next-elem` to explain `data`. Computes the effective containers from an input object, as well as the display and debug outputs.

Parameters

```
elem-of-container(
  data: opaque,
  obj: container
) -> elem
```

data opaque

Internal state.

obj container

Container to segment.

add-auto-margin

Applies an element's margin to itself

Parameters

`add-auto-margin(elem) -> elem`

next-elem

This function is reentering, allowing interactive computation of the layout. Given its internal state `data`, `next-elem` uses the helper functions `elem-of-placed` and `elem-of-container` to compute the dimensions of the next element, which may be an obstacle or a container.

Parameters

`next-elem(data: opaque) -> (elem, opaque)`

data opaque

Internal state, stores

- `size` the available page dimensions,
- `elems` the remaining elements to handle in reverse order (they will be popped),
- `obstacles` the running accumulator of previous obstacles;

push-elem

Updates the internal state to include the newly created element.

Parameters

`push-elem(`
 `data: opaque ,`
 `elem: elem`
`) -> opaque`

data opaque

Internal state.

elem elem

Element to register.

separate

Splits the input sequence into pages of elements (either obstacles or containers), and flowing content.

An “obstacle” is data produced by the `placed` function. It can contain arbitrary content, and defines a zone where flowing content cannot be placed.

A “container” is produced by the function `container`. It defines a region where (once the obstacles are subtracted) is allowed to contain flowing content.

Lastly flowing content is produced by the function `content`. It will be threaded through every available container in order.

```
#separate({
  // This is an obstacle
  placed(top + left, box(width: 50pt, height: 50pt))
  // This is a container
  container(height: 50%)
  // This is flowing content
  content[#lorem(50)]
})
```

Parameters

```
separate(seq: seq) -> (pages: array, flow: (..content,))
```

seq seq

A sequence of constructors `placed`, `container`, and `content`.

regions

Debug version of the toplevel `reflow`, that only displays the partitioned layout.

Parameters

```
regions(
  seq: seq,
  display: bool,
  placement
) -> content
```

seq seq

Input sequence to segment. Constructed from `placed`, `container`, and `content`.

display bool

Whether to show the placed objects (`true`), or only their hitbox (`false`).

Default: `true`

placement

Controls relation to other content on the page. See analogous `placement` option on `reflow`.

Default: `page`

VII.c Contouring (contour.typ)

Image boundary transformers.

- `margin()`

- [frac-rect\(\)](#)
- [horiz\(\)](#)
- [vert\(\)](#)
- [width\(\)](#)
- [height\(\)](#)
- [grid\(\)](#)
- [ascii-art\(\)](#)

Variables

- [phantom](#)

margin

Contouring function that pads the inner image.

Parameters

```
margin(..args) -> function
```

..args

May contain the following parameters, ordered here by decreasing generality and increasing precedence

- length for all sides, the only possible positional argument
- x,y: length for horizontal and vertical margins respectively
- top,bottom,left,right: length for single-sided margins

frac-rect

Helper function to turn a fractional box into an absolute one.

Parameters

```
frac-rect(  
  frac: (x: fraction, y: fraction, width: fraction, height: fraction) ,  
  abs: (x: length, y: length, width: length, height: length) ,  
  ..style  
) -> (x: length, y: length, width: length, height: length)
```

frac (x: fraction, y: fraction, width: fraction, height: fraction)

Child dimensions as fractions.

abs (x: length, y: length, width: length, height: length)

Parent dimensions as absolute lengths.

..style

Currently ignored.

horiz

Horizontal segmentation as (left, right)

Parameters

```
horiz(  
  div: int,  
  fun: function(fraction) => (fraction, fraction)  
) -> function
```

div int

Number of subdivisions.

Default: 5

fun function(fraction) => (fraction, fraction)

For each location, returns the left and right bounds.

vert

Vertical segmentation as (top, bottom)

Parameters

```
vert(  
  div: int,  
  fun: function(fraction) => (fraction, fraction)  
) -> function
```

div int

Number of subdivisions.

Default: 5

fun function(fraction) => (fraction, fraction)

For each location, returns the top and bottom bounds.

width

Horizontal segmentation as (anchor, width).

Parameters

```
width(  
  div: int,  
  flush: alignment,  
  fun: function(fraction) => (fraction, fraction)  
) -> function
```

div int

Number of subdivisions.

Default: 5

flush `alignment`

Relative horizontal alignment of the anchor.

Default: `center`

fun `function(fraction) => (fraction, fraction)`

For each location, returns the position of the anchor and the width.

height

Vertical segmentation as `(anchor, height)`.

Parameters

```
height(  
  div: int,  
  flush: alignment.,  
  fun: function(fraction) => (fraction, fraction)  
) -> function
```

div `int`

Number of subdivisions.

Default: `5`

flush `alignment.`

Relative vertical alignment of the anchor.

Default: `horizon`

fun `function(fraction) => (fraction, fraction)`

For each location, returns the position of the anchor and the height.

grid

Cuts the image into a rectangular grid then checks for each cell if it should be included. The resulting cells are automatically grouped horizontally.

Parameters

```
grid(  
  div: int (x: int, y: int),  
  fun: function(fraction, fraction) => bool  
) -> function
```

div `int` or `(x: int, y: int)`

Number of subdivisions.

Default: `5`

```
fun function(fraction, fraction) => bool
```

Returns for each cell whether it satisfies the 2D equations of the image's boundary.

ascii-art

Allows drawing the shape of the image as ascii art.

Blocks

- #: full
- : empty

Half blocks

- [: left
-]: right
- ^: top
- _: bottom

Quarter blocks

- `: top left
- ': top right
- ,: bottom left
- .: bottom right

Anti-quarter blocks

- J: top left
- L: top right
- 7: bottom left
- F: bottom right

Diagonals

- /: positive
- \: negative

Parameters

```
ascii-art(ascii: code)
```

```
ascii    code
```

Draw the shape of the image in ascii art.

phantom function

Drops all boundaries. Using `boundary: phantom` will let other content flow over this object.

VII.d Bisection (`bisect.typ`)

Content splitting algorithm.

- `fits-inside()`
- `default-rebuild()`
- `take-it-or-leave-it()`
- `has-text()`
- `has-child()`
- `has-children()`
- `is-list-item()`

- [is-enum-item\(\)](#)
- [has-body\(\)](#)
- [dispatch\(\)](#)
- [fill-box\(\)](#)

fits-inside

Tests if content fits inside a box.

WARNING: horizontal fit is not very strictly checked A single word may be said to fit in a box that is less wide than the word. This is an inherent limitation of `measure(box(...))` and I will try to develop workarounds for future versions.

The closure of this function constitutes the basis of the entire content splitting algorithm: iteratively add content until it no longer `fits-inside`, with what “iteratively add content” means being defined by the content structure. Essentially all remaining functions in this file are about defining content that can be split and the correct way to invoke `fits-inside` on them.

```
#let dims = (width: 100%, height: 50%)
#box(width: 7cm, height: 3cm)[#layout(size =>
context {
  let words = [#lorem(12)]
  [#fits-inside(dims, words, size: size)]
  linebreak()
  box(..dims, stroke: 0.1pt, words)
}]]
```

true

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor.

```
#let dims = (width: 100%, height: 50%)
#box(width: 7cm, height: 3cm)[#layout(size =>
context {
  let words = [#lorem(15)]
  [#fits-inside(dims, words, size: size)]
  linebreak()
  box(..dims, stroke: 0.1pt, words)
}]]
```

false

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor
incididunt ut labore.

Parameters

```
fits-inside(
  dims: (width: relative, height: relative),
  ct: content,
  size: (width: length, height: length)
) -> bool
```

dims (width: relative, height: relative)

Maximum container dimensions. Relative lengths are allowed.

ct content

Content to fit in.

size (width: length, height: length)

Dimensions of the parent container to resolve relative sizes. These must be absolute sizes.

Default: `none`

default-rebuild

Destructure and rebuild content, separating the outer content builder from the rest to allow substituting the inner contents. In practice what we will usually do is recursively split the inner contents and rebuild the left and right halves separately.

Inspired by wrap-it's implementation (see: `_rewrap` in [github:ntjess/wrap-it](https://github.com/ntjess/wrap-it))

```
#let content = box(stroke: red)[Initial]
#let (inner, rebuild) = default-rebuild(
  content, "body",
)

Content: #content \
Inner: #inner \
Rebuild: #rebuild("foo")
```

```
#let content = [*_Initial_*]
#let (inner, rebuild) = default-rebuild(
  content, "body",
)

Content: #content \
Inner: #inner \
Rebuild: #rebuild("foo")
```

```
#let content = [a:b]
#let (inner, rebuild) = default-rebuild(
  content, "children",
)

Content: #content \
Inner: #inner \
Rebuild: #rebuild(([x], [y]))
```

Content: **Initial**
Inner: Initial
Rebuild: **foo**

Content: **Initial**
Inner: *Initial*
Rebuild: **foo**

Content: **a:b**
Inner: ([a], [:], [b])
Rebuild: **xy**

Parameters

```
default-rebuild(
  ct: content,
  inner-field: string
) -> (dictionnary, function)
```

inner-field string

What “inner” field to fetch (e.g. “body”, “text”, “children”, etc.)

take-it-or-leave-it

“Split” opaque content.

Parameters

```
take-it-or-leave-it(
  ct: content,
  fits-inside: function
) -> (content?, content?)
```

ct content

This content cannot be split. If it fits take it, otherwise keep it for later.

fits-inside `function`

Closure to determine if the content fits (see `fits-inside` above).

has-text

Split content with a "text" main field. Strategy: split by " " and take all words that fit. Then if hyphenation is enabled, split by syllables and take all syllables that fit. End the block with a linebreak that has the justification of the paragraph.

Parameters

```
has-text(  
  ct: content,  
  split-dispatch: function,  
  fits-inside: function,  
  cfg: dictionary  
)
```

`ct` `content`

Content to split.

`split-dispatch` `function`

Recursively passed around (see `split-dispatch` below).

`fits-inside` `function`

Closure to determine if the content fits (see `fits-inside` above).

`cfg` `dictionary`

Extra configuration options.

has-child

Split content with a "child" main field. Strategy: recursively split the child.

Parameters

```
has-child(  
  ct: content,  
  split-dispatch: function,  
  fits-inside: function,  
  cfg: dictionary  
)
```

`ct` `content`

Content to split.

`split-dispatch` `function`

Recursively passed around (see `split-dispatch` below).

fits-inside `function`

Closure to determine if the content fits (see `fits-inside` above).

cfg `dictionary`

Extra configuration options.

has-children

Split content with a "children" main field. Strategy: take all children that fit.

Parameters

```
has-children(  
  ct: content,  
  split-dispatch: function,  
  fits-inside: function,  
  cfg: dictionary  
)
```

ct `content`

Content to split.

split-dispatch `function`

Recursively passed around (see `split-dispatch` below).

fits-inside `function`

Closure to determine if the content fits (see `fits-inside` above).

cfg `dictionary`

Extra configuration options.

is-list-item

Split a `list.item`. Strategy: recursively split the body, and do some magic to simulate a bullet point indent.

Parameters

```
is-list-item(  
  ct: content,  
  split-dispatch: function,  
  fits-inside: function,  
  cfg: dictionary  
)
```

ct `content`

Content to split.

split-dispatch function

Recursively passed around (see `split-dispatch` below).

fits-inside function

Closure to determine if the content fits (see `fits-inside` above).

cfg dictionary

Extra configuration options.

is-enum-item

Split an `enum.item`. Strategy: recursively split the body, and do some magic to simulate a numbering indent.

Parameters

```
is-enum-item(  
  ct: content,  
  split-dispatch: function,  
  fits-inside: function,  
  cfg: dictionary  
)
```

ct content

Content to split.

split-dispatch function

Recursively passed around (see `split-dispatch` below).

fits-inside function

Closure to determine if the content fits (see `fits-inside` above).

cfg dictionary

Extra configuration options.

has-body

Split content with a "body" main field. There is a special strategy for `list.item` and `enum.item` which are handled separately. Elements `strong`, `emph`, `underline`, `stroke`, `overline`, `highlight` are splittable, the rest are treated as non-splittable.

Parameters

```
has-body(  
  ct: content,  
  split-dispatch: function,  
  fits-inside: function,  
  cfg: dictionary  
)
```

ct content

Content to split.

split-dispatch function

Recursively passed around (see `split-dispatch` below).

fits-inside function

Closure to determine if the content fits (see `fits-inside` above).

cfg dictionary

Extra configuration options.

dispatch

Based on the fields on the content, call the appropriate splitting function. This function is involved in a mutual recursion loop, which is why all other splitting functions take this one as a parameter.

Parameters

```
dispatch(  
  ct: content,  
  fits-inside: function,  
  cfg: dictionary  
)
```

ct content

Content to split.

fits-inside function

Closure to determine if the content fits (see `fits-inside` above).

cfg dictionary

Extra configuration options.

fill-box

Initialize default configuration options and take as much content as fits in a box of given size. Returns a tuple of the content that fits and the content that overflows separated.

Parameters

```
fill-box(  
  dims: (width: length, height: length) ,  
  ct: content ,  
  size: (width: length, height: length) ,  
  cfg: dictionary  
) -> (content, content)
```

dims (width: length, height: length)

Container size.

ct content

Content to split.

size (width: length, height: length)

Parent container size.

Default: none

cfg dictionary

Configuration options.

- **list-markers**: (..content,), default value ([•], [‣], [-], [•], [‣], [-]). If you change the markers of list, put the new value in the parameters so that lists are correctly split.
- **enum-numbering**: (..str,), default value ("1.", "1.", "1.", "1.", "1.", "1."). If you change the numbering style of enum, put the new style in the parameters so that enums are correctly split.

Default: (:)

VII.e Threading (threading.typ)

Filling and stretches boxes iteratively.

- [smart-fill-boxes\(\)](#)
- [reflow\(\)](#)

smart-fill-boxes

Thread text through a list of boxes in order, allowing the boxes to stretch vertically to accomodate for uneven tiling.

Parameters

```
smart-fill-boxes(  
  body: content ,  
  avoid: (..block,) ,  
  boxes: (..block,) ,  
  extend: length ,  
  size: (width: length, height: length)  
) -> (..content,)
```

body content

Flowing text.

```
avoid  (.block,)
```

Obstacles to avoid. A list of (x: length, y: length, width: length, height: length).

Default: ()

```
boxes  (.block,)
```

Boxes to fill. A list of (x: length, y: length, width: length, height: length, bound: block).

bound is the upper limit of how much to stretch the container, i.e. also (x: length, y: length, width: length, height: length).

Default: ()

```
extend  length
```

How much the baseline can extend downwards (within the limits of bounds).

Default: 1em

```
size  (width: length, height: length)
```

Dimensions of the container as given by layout.

Default: none

reflow

Segment the input sequence according to the tiling algorithm, then thread the flowing text through it.

Parameters

```
reflow(  
  seq: seq,  
  debug: bool,  
  overflow: any,  
  placement  
) -> content
```

```
seq  seq
```

See module tiling for how to format this content.

```
debug  bool
```

Whether to show the boundaries of boxes.

Default: false

overflow any

Controls the behavior in case the content overflows the provided containers.

- `false` -> adds a warning box to the document
- `true` -> ignores any overflow
- `pagebreak` -> the text that overflows is simply placed normally on the next page
- `panic` -> refuses to compile the document
- any content => content function -> uses that for formatting

Default: `false`

placement

Relationship with the rest of the content on the page.

- `page`: content is not visible to the rest of the layout, and will be placed at the current location. Supports pagebreaks.
- `box`: meander will simulate a box of the same dimensions as its contents so that normal text can go before and after. Supports pagebreaks.
- `float`: similar to `page` in that it is invisible to the rest of the content, but always placed at the top left of the page. Does not support pagebreaks.

Default: `page`