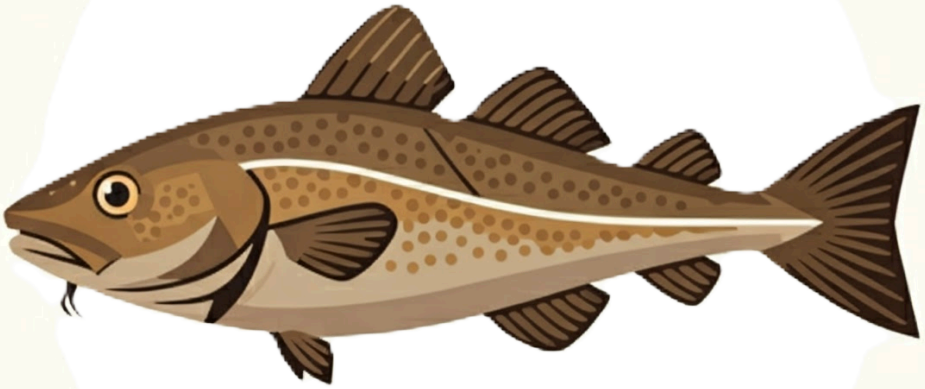
Always start a new Typst project by importing codly



Codly v1.1.0 manual

Your code blocks on steroids

O RLY?

Dherse

Contents

1. Codly

Codly is a library that enhances the way you write code blocks in Typst. It provides a set of tools to help you manage your code blocks, highlights them, skip parts of them, and more. This manual will guide you through the different features of Codly, how to use them, and how to integrate them into your Typst projects.

> **Notification**
>
> If you find any issues with Codly, please report them on the GitHub repository: https://github.com/Dherse/codly.

1.1. Initialization

To start using Codly, you must first import it into your Typst project.

Example code

```
#import "@preview/codly:1.1.0": *

#show: codly-init
```

Rendered output

As you can see, this does nothing but initialize codly. You can also import it with a specific version, as shown in the example above. For the latest version, always refer to the Typst Universe page.

From this point on, any code block that is included in your Typst project will be enhanced by Codly.

Example code Typst

````
```
Hello, world!
```
````

Rendered output

```
Hello, world!
```

1.2. Enabling and disabling codly

By default Codly will be enabled after initialization. However, disabling codly can be done using the `codly-disable` function, the `enabled` argument of the `codly` function, or the `no-codly` function. To enable Codly again, use the `codly-enable` function or by setting the `enabled` parameter again.

2. A primer on Codly's show-rule like system

Codly uses a function called `codly` to create a kind of show-rule which you can use to configure how your code blocks are displayed. The `codly` function takes a set of arguments that define how the code block should be displayed. Here is the equivalent definition of the `codly` function:

Typst code

```
let codly(
  enabled: true,
  offset: 0,
  offset-from: none,
  range: none,
  ranges: (),
  languages: (:),
  display-name: true,
  display-icon: true,
  default-color: rgb("#283593"),
  radius: 0.32em,
  inset: 0.32em,
  fill: none,
  zebra-fill: luma(240),
  stroke: 1pt + luma(240),
  lang-inset: 0.32em,
  lang-outset: (x: 0.32em, y: 0pt),
  lang-radius: 0.32em,
  lang-stroke: (lang) => lang.color + 0.5pt,
  lang-fill: (lang) => lang.color.lighten(80%),
  lang-format: codly.default-language-block,
  number-format: (number) => [ #number ],
  number-align: left + horizon,
  smart-indent: false,
  annotations: none,
  annotation-format: numbering.with("(1)"),
  highlights: none,
  highlight-radius: 0.32em,
  highlight-fill: (color) => color.lighten(80%),
  highlight-stroke: (color) => 0.5pt + color,
  highlight-inset: 0.32em,
  reference-by: line,
  reference-sep: "-",
  reference-number-format: numbering.with("1"),
  header: none,
  header-repeat: false,
  header-transform: (x) => x,
  header-cell-args: (),
  footer: none,
  footer-repeat: false,
  footer-transform: (x) => x,
  footer-cell-args: (),
  breakable: false,
) = {}
```

The codly functions acts like a set-rule, this means that calling it will set the configuration for all code blocks that follow it, with the exception of a few arguments that are explicitly set for each code block. To perform changes locally, you can use the `local` function, or set the arguments before the code block and reset them after to their previous values.

> ⚠ **Warning**
>
> Unlike regular set-rules in native Typst, there are two considerations:
> - The `codly` function uses states to store the configuration, this means that it is dependent on layout order for the order in which settings are applied.
> - The `codly` function is not local, it sets the configuration for all code blocks that follow it in layout order, unless overriden by another `codly` call. This means that you cannot use it to set the configuration for a specific code block. To perform this, use the `local` function to set the configuration for a specific "section".

2.1. Enabled (`enabled`)

Type	`bool`
Default value	`true`
Contextual function	✓ yes
Automatically reset	✗ no

Whether codly is enabled or not. If it is disabled, the code block will be displayed as a normal code block, without any additional codly-specific formatting. This is useful if you want to disable codly for a specific block. You can also disable codly locally using the `no-codly` function, or disable it and enable it again using the `codly-disable` and `codly-enable` functions.

2.1.1. Example

Example code (Typst)

```
*Enabled = true*:
#codly(enabled: true)
```typ
Hello, world!
```

Enabled = false:
#codly(enabled: false)
```typ
Hello, world!
```
```

Rendered output

Enabled = true:

```typ
Hello, world!
```

Enabled = false:

Hello, world!

2.2. Header (`header`)

Type	`content` or `none`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✓ yes

An optional header to display above the code block. It can be optionally repeated on all subsequent pages with the `header-repeat` argument. And additional customizations are available with the `header-cell-args` and `header-transform` arguments.

2.2.1. Example

Example code (Typst)

```
#codly(header: [*Hello, world!*])
```typ
Hello, world!
```
```

Rendered output

Hello, world!

```typ
Hello, world!
```

2.3. Header Repeat (`header-repeat`)

Type	`bool`
Default value	`false`
Contextual function	✓ yes
Automatically reset	✗ no

Whether to repeat the header on each page. This is only applicable if a header is provided, if the code block is `breakable`, and if it actually breaks on more than one page. For more information see `grid.header:repeat`.

2.4. Header Cell Args (`header-cell-args`)

Type	`array`, `dictionary`, or `arguments`
Default value	`()`
Contextual function	✓ yes
Automatically reset	✗ no

Additional arguments to be provided to the `grid.cell` containing the header. Lets you customize the header cell further. Internally, codly wraps the content of the `header` argument in a `grid.cell` with these arguments. The only argument that is always common is the `body` argument which is the value of the `header` argument, and the `colspan` which is always set to `2`.

For a full description of the argument, look at the documentation of the `grid.cell` function.

2.4.1. Example

Example code Typst

```
1 //Centering the header:
2 #codly(
3   header: [*Hello, world!*],
4   header-cell-args: (align: center, )
5 )
6
7 ```typ
8 Hello, world!
9 ```
```

Rendered output

Hello, world! typ

```
1 Hello, world!
```

2.5. Header Transform (`header-transform`)

Type	function
Default value	`(x) => x`
Contextual function	✗ no
Automatically reset	✗ no

Function that transforms the header into arbitrary content to be stored in the `grid.cell`. Can be seen as a show-rule for the header. This allows to perform global transformation/show-rule like operations on the header.

2.5.1. Example

Example code (Typst)

```typ
//Making the header bold and blue:
#codly(
  header: [Hello, world!],
  header-transform: (x) => {
    set text(fill: blue)
    strong(x)
  }
)

```typ
Hello, world!
```
```

Rendered output

Hello, world! (typ)

```
Hello, world!
```

2.6. Footer (`footer`)

Type	content or none
Default value	none
Contextual function	✓ yes
Automatically reset	✓ yes

An optional footer to display below the code block. See header for more information.

2.6.1. Example

Example code (Typst)

```typ
#codly(footer: [*Hello, world!*])
```typ
Hello, world!
```
```

Rendered output

```
Hello, world!
```
(typ)

Hello, world!

2.7. Footer Repeat (`footer-repeat`)

Type	bool
Default value	false
Contextual function	✓ yes
Automatically reset	✗ no

Whether to repeat the footer on each page. See header-repeat for more information.

2.8. Footer Cell Args (`footer-cell-args`)

Type	`array` , `dictionary` , or `arguments`
Default value	`()`
Contextual function	✓ yes
Automatically reset	✗ no

Additional arguments to be provided to the `grid.cell` containing the footer. See `header-cell-args` for more information.

2.8.1. Example

Example code (Typst)

```
//Centering the footer:
#codly(
 footer: [*Hello, world!*],
 footer-cell-args: (align: center, )
)

```typ
Hello, world!
```
```

Rendered output

```
Hello, world!
```

Hello, world!

2.9. Footer Transform (`footer-transform`)

Type	`function`
Default value	`(x) => x`
Contextual function	✗ no
Automatically reset	✗ no

Function that transforms the footer into arbitrary content to be stored in the `grid.cell` . Can be seen as a show-rule for the footer. See `header-transform` for more information.

2.9.1. Example

Example code (Typst)

```
//Making the footer bold and blue:
#codly(
   footer: [Hello, world!],
   footer-transform: (x) => {
     set text(fill: blue)
     strong(x)
   }
)

```typ
Hello, world!
```
```

Rendered output

```
Hello, world!
```

Hello, world!

2.10. Offset (`offset`)

Type	`int`
Default value	`0`
Contextual function	✓ yes
Automatically reset	✓ yes

The offset to apply to line numbers.

This is purely cosmetic, only impacting the shown line numbers in the final output.

2.10.1. Example

Example code Typst

```
*No offset:*
```typ
Hello, world!
```

Offset by 5:
#codly(offset: 5)
```typ
Hello, world!
```
```

Rendered output

No offset:

```
1 Hello, world!
```
typ

Offset by 5:

```
6 Hello, world!
```
typ

2.11. Offset from other code block (`offset-from`)

Type	`none` or `label`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✓ yes

The offset to apply to line numbers, relative to another code block. This is useful when you want to match line numbers between two code blocks. This code block will continue the line numbers from the other code block, with the specified offset.

This is done by giving a `label` to the parent raw block, and then setting it as the `offset-from` on the second code block.

> **Info**
>
> Note that the offset obtained from the other code block is added to the offset specified in the `offset` argument.

> **Warning**
>
> **Important**: this feature works with any `offset` set on the other code block, including `offset-from` but may give unexpected results if both code blocks have `offset-from` set to each other or if the preceeding code block has `range` or `skips` set.

> **Experiment**
>
> This feature should be considered experimental. Please report any issues you encounter on GitHub: https://github.com/Dherse/codly.

2.11.1. Example

Example code Typst

````
```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
``` <fib-fn>

*Will continue at line 5*
#codly(offset-from: <fib-fn>)
```py
fib(25)
```
````

Rendered output

```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
```

Will continue at line 5

```py
fib(25)
```

2.12. Range (`range`)

Type	`none` or `array`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✓ yes

The range of line numbers to display, zero-indexed. If set to `none`, all lines are displayed. Can also be achieved using the convenience function `codly-range`. If set to `none`, all lines are displayed.

2.12.1. Example

Example code (Typst)

```
#codly(range: (2, 4))
```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```py
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
```

2.13. Ranges (`ranges`)

Type	`none` or `array`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✓ yes

The ranges of line numbers to display, zero-indexed. If set to `none`, all lines are displayed. Can also be achieved using the convenience function `codly-range` if provided with more than one range. If set to `none`, all lines are displayed.

> **Info**
>
> Overrides the `range` argument.

2.13.1. Example

Example code (Typst)

```
#codly(ranges: ((2, 2), (4, 4)))
```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```
  if n <= 1:
  return fib(n - 1) + fib(n - 2)
```

2.14. Languages (`languages`)

Type	dictionary
Default value	`(:)`
Contextual function	✓ yes
Automatically reset	✗ no

The language definitions to use for language block formatting. It is defined as a dictionary where the keys are the language names andeach value is another dictionary containing the following keys:

- `name` : the “pretty” name of the language as a content/showable value
- `color` : the color of the language, if omitted uses the default color
- `icon` : the icon of the language, if omitted no icon is shown.

If an entry is missing, and language blocks are enabled, will show the “un-prettified” language name, with the default color.

2.14.1. Example

Example code (Typst)

```typst
#codly(
  languages: (
    py: (
      name: [Python], color: green, icon: "🐍"
    ),
  )
)
```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```python
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```

2.14.2. Pre-existing language definitions

> **Info**
>
> Check out the `codly-languages` package on Typst universe. It contains pre-definition for many language and is extremely easy to use. You can consider it officially endorsed by the codly author as of the 19th of November 2024.

Example code (Typst)

```typst
#import "@preview/codly-languages:0.1.1": *
#codly(languages: codly-languages)
```rust
fn main() {
  println!("Hello, world!");
}
```
```zig
const std = @import("std");

pub fn main() void {
  std.debug.print("Hello, world!", .{});
}
```
```

Rendered output

```rust
fn main() {
  println!("Hello, world!");
}
```

```zig
const std = @import("std");

pub fn main() void {
  std.debug.print("Hello, world!", .{});
}
```

2.15. Default language color (`default-color`)

Type	`color` , `gradient` , or `pattern`
Default value	`rgb("#283593")`
Contextual function	✓ yes
Automatically reset	✗ no

The default color to use for language blocks. Used when a language is not defined in the `languages` argument. Also note that it is only used when the `lang-format` is its `auto` or you are using it in a custom formatter. If you are using a custom formatter, it is passed to the formatter as a named argument `color` .

2.15.1. Example

Example code (Typst)

```
*Default color:*
```py
print('Hello, world!')
print('Hello, world!')
```
Overriden default color:
#codly(default-color: orange)
```py
print('Hello, world!')
print('Hello, world!')
```
```

Rendered output

Default color:

```py
print('Hello, world!')
print('Hello, world!')
```

Overriden default color:

```py
print('Hello, world!')
print('Hello, world!')
```

2.16. Radius (`radius`)

Type	`length`
Default value	`0.32em`
Contextual function	✓ yes
Automatically reset	✗ no

The radius of the border of the code block, see `block.radius` for more information.

2.16.1. Example

Example code (Typst)

```
*Default radius:*
```py
print('Hello, world!')
print('Hello, world!')
```
Overriden radius:
#codly(radius: 2em)
```py
print('Hello, world!')
print('Hello, world!')
```
Zero radius:
#codly(radius: 0pt)
```py
print('Hello, world!')
print('Hello, world!')
```
```

Rendered output

Default radius:

```py
print('Hello, world!')
print('Hello, world!')
```

Overriden radius:

```py
print('Hello, world!')
print('Hello, world!')
```

Zero radius:

```py
print('Hello, world!')
print('Hello, world!')
```

2.17. Inset (`inset`)

Type	`length` or `dictionary`
Default value	`0.32em`
Contextual function	✓ yes
Automatically reset	✗ no

Inset of the code lines, that is the distance between the border and the code lines. It can also be a dictionary with the keys same keys as in the Tyspt built-in `block.inset` .

2.17.1. Example

Example code (Typst)

```
*Default inset:*
```py
print('Hello, world!')
```
Overriden inset:
#codly(inset: 1em)
```py
print('Hello, world!')
```
```

Rendered output

Default inset:

```py
print('Hello, world!')
```

Overriden inset:

```py
    print('Hello, world!')
```

2.17.2. Example: Dictionary inset

Example code (Typst)

```
*Default inset:*
```py
print('Hello, world!')
```
Overriden inset:
#codly(inset: (x: 0.32em, y: 0.1em))
```py
print('Hello, world!')
```
```

Rendered output

Default inset:

```py
print('Hello, world!')
```

Overriden inset:

```py
print('Hello, world!')
```

2.18. Fill (`fill`)

Type	`none` , `color` , `gradient` , or `pattern`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✗ no

The fill of the code block when not zebra-striped.

2.18.1. Example

Example code Typst

```
*Default fill:*
```py
print('Hello, world!')
print('Hello, world!')
```
Overriden fill:
#codly(fill:
gradient.linear(..color.map.flare))
```py
print('Hello, world!')
print('Hello, world!')
```
No fill:
#codly(fill: none)
```py
print('Hello, world!')
print('Hello, world!')
```
```

Rendered output

Default fill:

```py
print('Hello, world!')
print('Hello, world!')
```

Overriden fill:

```py
print('Hello, world!')
print('Hello, world!')
```

No fill:

```py
print('Hello, world!')
print('Hello, world!')
```

2.19. Zebra fill (`zebra-fill`)

Type	`none` , `color` , `gradient` , or `pattern`
Default value	`luma(240)`
Contextual function	✓ yes
Automatically reset	✗ no

Background color of the code lines when zebra-stripped. If set to `none` , no zebra-striping is applied.

2.19.1. Example

Example code (Typst)

```
*Default zebra:*
```py
print('Hello, world!')
print('Hello, world!')
```
No zebra:
#codly(zebra-fill: none)
```py
print('Hello, world!')
print('Hello, world!')
```
Overriden zebra:
#codly(zebra-fill:
gradient.linear(..color.map.flare))
```py
print('Hello, world!')
```
```

Rendered output

Default zebra:

```py
print('Hello, world!')
print('Hello, world!')
```

No zebra:

```py
print('Hello, world!')
print('Hello, world!')
```

Overriden zebra:

```py
print('Hello, world!')
```

2.20. Stroke (`stroke`)

Type	`none` or `stroke`
Default value	`1pt + luma(240)`
Contextual function	✓ yes
Automatically reset	✗ no

The stroke to surround the whole code block with?

2.20.1. Example

Example code (Typst)

```
*Default stroke:*
```py
print('Hello, world!')
```
No stroke:
#codly(stroke: none)
```py
print('Hello, world!')
```
Overriden stroke:
#codly(stroke: 1pt + blue)
```py
print('Hello, world!')
```
```

Rendered output

Default stroke:

```py
print('Hello, world!')
```

No stroke:

```py
print('Hello, world!')
```

Overriden stroke:

```py
print('Hello, world!')
```

2.21. Language box inset (`lang-inset`)

Type	`length` or `dictionary`
Default value	`0.32em`
Contextual function	✓ yes
Automatically reset	✗ no

The inset of the language block. This only applies if you're using the default language block formatter. It can also be a dictionary with the keys same keys as in the Tyspt built-in `block.inset`

2.21.1. Example

Example code Typst

```
#codly(lang-inset: 5pt)
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

```py
print('Hello, world!')
print('Goodbye, world!')
```

2.22. Language box outset (`lang-outset`)

Type	`dictionary`
Default value	`(x: 0.32em, y: 0em)`
Contextual function	✓ yes
Automatically reset	✗ no

The X and Y outset of the language block, applied as a `dx` and `dy` during the `place` operation. This applies in every case, whether or not you're using the default language block formatter. The default value is chosen to get rid of the `inset` applied to each line.

2.22.1. Example

Example code Typst

```
#codly(lang-outset: (x: -10pt, y: 5pt))
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

```py
print('Hello, world!')
print('Goodbye, world!')
```

2.23. Language box radius (`lang-radius`)

Type	`length`
Default value	`0.32em`
Contextual function	✓ yes
Automatically reset	✗ no

The radius of the border of the language block.

2.23.1. Example

Example code Typst

```
#codly(lang-radius: 10pt)
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

```py
print('Hello, world!')
print('Goodbye, world!')
```

2.24. Language box stroke (`lang-stroke`)

Type	`none` , `stroke` , or `function`
Default value	`(lang) => lang.color + 0.5pt`
Contextual function	× no
Automatically reset	× no

The stroke of the language block. Can be a function that takes in the language `dictionary` or `none` (see argument `languages`) and returns a stroke.

2.24.1. Example

Example code (Typst)

```
*Fixed stroke:*
#codly(lang-stroke: 1pt + red)
```py
print('Hello, world!')
print('Goodbye, world!')
```
Function mapping:
#codly(lang-stroke: (lang) => 2pt +
lang.color)
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

Fixed stroke:

```py
print('Hello, world!')
print('Goodbye, world!')
```

Function mapping:

```py
print('Hello, world!')
print('Goodbye, world!')
```

2.25. Language box fill (`lang-fill`)

Type	`none` , `color` , `gradient` , `pattern` , or `function`
Default value	`(lang) => lang.color.lighten(80%)`
Contextual function	× no
Automatically reset	× no

The background color of the language block. Can be a function that takes in the language `dictionary` or `none` (see argument `languages`) and returns a stroke.

2.25.1. Example

Example code (Typst)

```
*Fixed fill:*
#codly(lang-fill: red)
```py
print('Hello, world!')
print('Goodbye, world!')
```
Function mapping:
#codly(lang-fill: (lang) =>
lang.color.lighten(40%))
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

Fixed fill:

```py
print('Hello, world!')
print('Goodbye, world!')
```

Function mapping:

```py
print('Hello, world!')
print('Goodbye, world!')
```

2.26. Language box formatter (`lang-format`)

Type	`type(auto)` , `none` , or `function`
Default value	`auto`
Contextual function	✕ no
Automatically reset	✕ no

The formatter for the language block. A value of `none` will not display the language block. To use the default formatter, set to `auto` . The function takes three arguments:

- `lang` : the language key (e.g. `py`)
- `icon` : the language icon, can be none or empty content
- `color` : the language color

The function should return a content/showable value.

> **Info**
>
> The language formatter should avoid using `state` as this can lead to quadratic execution time, see typst/typst #5220 for more information. Internally, when set to `auto` , codly uses an inlined function to avoid using states.

2.26.1. Example

Example code (Typst)

```
*Default formatter:*
```py
print('Hello, world!')
```
Function mapping:
#codly(lang-format: (_, _, _) => [No!]))
```py
print('Hello, world!')
```
```

Rendered output

Default formatter:

```
1 print('Hello, world!')
```
py

Function mapping:)

```
1 print('Hello, world!')
```
No!

2.27. Display language name (`display-name`)

Type	bool
Default value	true
Contextual function	✓ yes
Automatically reset	✗ no

Whether to display the name of the language in the language block. This only applies if you're using the default language block formatter.

2.27.1. Example

Example code (Typst)

```
#codly(
  display-name: false,
  languages: (
    py: (
      name: [Python], color: green,
      icon: "🐍"
    ),
  ),
)
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

```py
print('Hello, world!')
print('Goodbye, world!')
```

2.28. Display language icon (`display-icon`)

Type	bool
Default value	true
Contextual function	✓ yes
Automatically reset	✗ no

Whether to display the icon of the language in the language block. This only applies if you're using the default language block formatter.

2.28.1. Example

Example code (Typst)

```
#codly(
  display-icon: false,
  languages: (
    py: (
      name: [Python], color: green,
      icon: "🐍"
    ),
  ),
)
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

```py
print('Hello, world!')
print('Goodbye, world!')
```

2.29. Line number format (`number-format`)

Type	function or none
Default value	`numbering.with("1")`
Contextual function	✗ no
Automatically reset	✗ no

The format of the line numbers, a function that takes in number and returns a content. If set to none, disables line numbers.

2.29.1. Example

Example code (Typst)

```
#codly(number-format: numbering.with("I."))
```py
print('Hello, world!')
print('Goodbye, world!')
```
```

Rendered output

```
I.  print('Hello, world!')
II. print('Goodbye, world!')
```

2.30. Line number alignment (`number-align`)

Type	alignment
Default value	`left + horizon`
Contextual function	✓ yes
Automatically reset	✗ no

The alignment of the numbers.

2.30.1. Example

Example code (Typst)

```
#codly(number-align: right + top)
```py
# Iterative Fibonacci
# As opposed to the recursive
# version
def fib(n):
  if n <= 1:
    return n
  last, current = 0, 1
  for _ in range(2, n + 1):
  last, current = current, last + current
  return current
fib(25)
```
```

Rendered output

```
 1 # Iterative Fibonacci
 2 # As opposed to the recursive
 3 # version
 4 def fib(n):
 5   if n <= 1:
 6     return n
 7   last, current = 0, 1
 8   for _ in range(2, n + 1):
 9   last, current = current, last + current
10   return current
11 fib(25)
```

2.31. Smart indentation (`smart-indent`)

Type	bool
Default value	true
Contextual function	× no
Automatically reset	× no

Whether to use smart indentation, which will check for indentation on a line and use a bigger left side inset instead of spaces. This allows for linebreaks to continue at the same level of indentation. This is on by default, but disabling it can improve performance.

2.31.1. Example

Example code Typst

````
*Enabled (default):*
```py
def quicksort(L):
  qsort = lambda L: [] if L==[] else qsort([x for x in L[1:] if x< L[0]]) + L[0:1] + qsort([x for x in L[1:] if x>=L[0]])
  qsort(L)
```
*Disabled:*
#codly(smart-indent: false)
```py
def quicksort(L):
  qsort = lambda L: [] if L==[] else qsort([x for x in L[1:] if x< L[0]]) + L[0:1] + qsort([x for x in L[1:] if x>=L[0]])
  qsort(L))
```
````

Rendered output

Enabled (default):

```py
def quicksort(L):
  qsort = lambda L: [] if L==[] else qsort([x for x in L[1:] if x< L[0]]) + L[0:1] + qsort([x for x in L[1:] if x>=L[0]])
  qsort(L)
```

Disabled:

```py
def quicksort(L):
  qsort = lambda L: [] if L==[] else qsort([x for x in L[1:] if x< L[0]]) + L[0:1] + qsort([x for x in L[1:] if x>=L[0]])
  qsort(L))
```

2.32. Breakable (`breakable`)

Type	bool
Default value	true
Contextual function	× no
Automatically reset	× no

Whether the codeblocks are breakable across page and column breaks.

2.33. Skips (`skips`)

Type	`array` or `none`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✓ yes

Insert a skip at the specified line numbers, setting its offset to the length of the skip. The skip is formatted using the `skip-number` argument. Each skip is an array with two values: the line where the skip is inserted (zero indexed) and the number of lines of the skip. The same behavior can be achieved using the `codly-skip` function.

2.33.1. Example

Example code (Typst)

```
#codly(skips: ((4, 32), ))
```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
…
fib(25)
```

2.34. Skip line (`skip-line`)

Type	`content` or `none`
Default value	`align(center)[ ... ]`
Contextual function	✓ yes
Automatically reset	✗ no

Sets the content with which the line code is filled when a skip is encountered.

2.34.1. Example

Example code (Typst)

```
#codly(
  skips: ((4, 32), ),
  skip-line: align(center,
  emoji.face.shock)
)
```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
…
😱
fib(25)
```

2.35. Skip number (`skip-number`)

Type	`content` or `none`
Default value	`[ ... ]`
Contextual function	✓ yes
Automatically reset	✗ no

Sets the content with which the line number columns is filled when a skip is encountered. If line numbers are disabled, this has no effect.

2.35.1. Example

Example code (Typst)

```
#codly(
  skips: ((4, 32), ),
  skip-number: align(center,
emoji.face.shock)
)
```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```py
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
😱                    …
fib(25)
```

2.36. Annotations (`annotations`)

Type	`array` or `none`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✓ yes

The annotations to display on the code block. A list of annotations that are automatically numbered and displayed on the right side of the code block.

Each entry is a dictionary with the following keys:

- `start` : the line number to start the annotation
- `end` : the line number to end the annotation, if missing or `none` the annotation will only contain the start line
  - `content` : the content of the annotation as a showable value, if missing or `none` the annotation will only contain the number
  - `label` : **if and only if** the code block is in a `figure` , sets the label by which the annotation can be referenced.

Generally you probably want the `content` to be contained within a `rotate(90deg)` .

Note: Annotations cannot overlap. **Known issues**:

- Annotations that spread over a page break will not work correctly
- Annotations on the first line of a code block will not work correctly.
- Annotations that span lines that overflow (one line of code two lines of text) will not work correctly.

> **Experiment**
>
> This feature should be considered experimental. Please report any issues you encounter on GitHub: https://github.com/Dherse/codly.

2.36.1. Example

Example code (Typst)

```
#codly(
  annotations:(
    (
      start: 1, end: 4,
      content: block(
        width: 2em,
        rotate(-90deg, reflow: true,
          align(center)[Function body])
      )
    ),
  ),
)
```py
def fib(n):
  if n <= 1:
    return n
  else:
    return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```py
def fib(n):
  if n <= 1:
    return n
  else:
    return fib(n - 1) + fib(n - 2)
fib(25)
```

(1) Function body

2.37. Annotation formatter (`annotation-format`)

Type	`none` or `function`
Default value	`numbering.with("(1)")`
Contextual function	✗ no
Automatically reset	✗ no

The format of the annotation number. Can be `none` or a function that formats the annotation number.

2.38. Highlights (`highlights`)

Type	`array` or `none`
Default value	`none`
Contextual function	✓ yes
Automatically reset	✗ no

You can apply highlights to the code block using the `highlights` argument. It consists of a list of dictionaries, each with the following keys:

- `line` : the line number to start highlighting
  - `start` : the character position to start highlighting, zero if omitted or `none`
- `end` : the character position to end highlighting, the end of the line if omitted or `none`
- `fill` : the fill of the highlight, defaults to the default color
- `tag` : an optional tag to be displayed alongside the highlight.
- `label` : **if and only if** the code block is in a `figure` , sets the label by which the highlight can be referenced.

As with other code block settings, annotations are reset after each code block.

2.38.1. Example

Example code Typst

```
#codly(highlights: (
    (line: 3, start: 2, end: none, fill: red),
    (line: 4, start: 13, end: 19, fill: green, tag: "(a)"),
    (line: 4, start: 26, fill: blue, tag: "(b)"),
))
```

```py
def fib(n):
  if n <= 1:
    return n
  else:
    return fib(n - 1) + fib(n - 2)
print(fib(25))
```

Rendered output

```py
def fib(n):
  if n <= 1:
    return n
  else:
    return fib(n - 1) (a) + fib(n - 2) (b)
print(fib(25))
```

2.39. Highlight radius (`highlight-radius`)

Type	`length`
Default value	`0.32em`
Contextual function	✓ yes
Automatically reset	✓ yes

The radius of the highlights.

2.40. Highlight fill (`highlight-fill`)

Type	`function`
Default value	`(color) => color.lighten(80%)`
Contextual function	✗ no
Automatically reset	✗ no

The fill transformer of the highlights, is a function that takes in the highlight color and returns a fill.

2.41. Highlight stroke (`highlight-stroke`)

Type	`stroke` or `function`
Default value	`(color) => 0.5pt + color`
Contextual function	✗ no
Automatically reset	✗ no

The stroke transformer of the highlights, is a function that takes in the highlight color and returns a stroke.

2.42. Highlight inset (`highlight-inset`)

Type	`length`
Default value	`0.32em`
Contextual function	✓ yes
Automatically reset	✗ no

The inset of the highlights.

2.43. Reference by (`reference-by`)

Type	`str`
Default value	`"line"`
Contextual function	✓ yes
Automatically reset	✗ no

The mode by which references are displayed. Two modes are available:
- `line` : references are displayed as line numbers
- `item` : references are displayed as items, i.e by the `tag` for highlights and `content` for annotations.

2.44. Reference separator (`reference-sep`)

Type	`str` or `content`
Default value	`"-"`
Contextual function	✓ yes
Automatically reset	✗ no

The separator to use when referencing highlights and annotations.

2.45. Reference number format (`reference-number-format`)

Type	`function`
Default value	`numbering.with("1")`
Contextual function	✕ no
Automatically reset	✕ no

The format of the reference number line number, only used if `reference-by` is set to `"line"`.

3. Referencing code blocks, highlights, and annotations

This section of the documentation will detail how you can use codly to reference: lines, highlights, and annotations in your code blocks. To do this, here are the requirements that must be met **for each code block**:

- ☐ Numbering of figures must be turned on: `set figure(numbering: ...)`.
- ☐ The code block must be contained within a raw figure: `figure(kind: raw)`.
- ☐ The figure must have a label of its own: `figure(...)[...] <my-label>`.

3.1. Shorthand line references

You can references lines directly, if you have set a label correctly, using the shorthand syntax `@my-label:1` to reference the second line (zero-based index) of the code block with the label `<my-label>`. It will always use the `reference-number-format` argument of the `codly` function to format the line number.

> **Experiment**
>
> You might notice that the second reference in the example below is formatted like a `link`. This is because it internally uses a `show ref: ..` show-rule which produces a link. This is a limitation of Typst and cannot be easily changed.

Example code (Typst)

```
1  #figure(
2    caption: "A code block with a label"
3  )[
…                    …
8  ] <my-label>
9
   I can reference my code block: @my-label.
10 But I can also reference a specific line of
   the label: @my-label:1.
```

Rendered output

```
1 = Example                                  typ
2 *Hello, world!*
```

Listing 1: A code block with a label

I can reference my code block: Listing 1. But I can also reference a specific line of the label: Listing 1-1.

3.2. Highlight references

You can also highlight by reference, to do this, you need to set a label for your highlight in the `highlights` argument of the `codly` function. You can then reference the highlight using the shorthand syntax `@my-highlight` to reference the highlight with the label `<my-highlight>`. There are two supported `reference-by` modes:

- `"line"`: references the line of the highlight
- `"item"`: references the tag of the highlight, this requires that the `tag` be set **for each highlight**.

Example code (Typst)

```
1  #codly(
2    highlights: ((line: 1, start: 2, end: 7,
     label: <hl-1>),
3  ))
…                    …
13 I can also reference a specific highlight
   by its label: @hl-1.
```

Rendered output

```
1 = Example                                  typ
2 *Hello, world!*
```

Listing 2: A code block with a label

I can also reference a specific highlight by its label: Listing 2-2.

And using `"item"` mode:

Example code (Typst)

```
1  #codly(
2    highlights: ((line: 1, start: 2, end: 7,
     label: <hl-2>, tag: [ Highlight ]), ),
3    reference-by: "item",
4  )
…                    …
14 I can also reference a specific highlight
   by its label: @hl-2.
```

Rendered output

```
1 = Example                                  typ
2 *Hello, world! Highlight*
```

Listing 3: A code block with a label

I can also reference a specific highlight by its label: Listing 3- Highlight.

4. Getting nice icons

This is a short, non-exhaustive guide on how to get nicer icons for the languages of your code blocks. In the documentation, codly makes use of tabler-icons to display the language icons. But a more general approach is the following:

1. Chose a font that contains icons, such as:
   - Tabler Icons
   - Font Awesome
   - Material Icons
   - Look on Google Fonts for more options
2. Download the font and put it in your project (if using the CLI, you need to set the `--font` argument)
3. Using your font selector, select the icon you wish to use
   - For example, the language icon in Tabler Icons is `ebbe` (the unicode value of the icon, which you can find in the documentation of the font)
   - Use the `text` function to display the icon in your document by setting the font, size, and the unicode value of the icon:

```
1 text(font: "tabler-icons" | Font name, size: 1em, "\u{ebbe | UTF-8 icon code}")   Typst code
```

4. You can store it the `languages` argument of the `codly` function to use it for all code blocks in your document:

Example code (Typst)

```
1 #let icon = text(font: "tabler-icons",
  size: 1em, "\u{ebbe}")
2 #codly(languages: (text: (icon: icon,
  name: "Text")))
3 ```text
4 Hello, world!
5 ```
```

Rendered output

```
1 Hello, world!   Text
```

5. Congrats, you now have fancy icons!
6. …
7. But you can notice that the baseline of the icon is wrong, I find that this is generally the case with tabler, you can set the baseline to `0.1em` in the icon to fix it:

Example code (Typst)

```
1 #let icon = text(font: "tabler-icons",
  size: 1em, "\u{ebbe}", baseline: 0.1em)
2 #codly(languages: (text: (icon: icon,
  name: "Text")))
3 ```text
4 Hello, world!
5 ```
```

Rendered output

```
1 Hello, world!   Text
```

4.1. Typst language icon (`typst-icon`)

Additionally, codly ships with language definitions for the Typst language. You can use the `typst-icon` function to get the Typst icon for your code blocks. This function takes no arguments and returns the proper settings for codly to use the Typst icon.

> **Info**
>
> You can use the `..` spread operator to spread it into your own `languages` dictionary.

Example code (Typst)

```
1 #codly(languages: typst-icon)
2 ```typ
3 = Here's a title
4 Hello, world!
5 ```
```

Rendered output

```
1 = Here's a title   Typst
2 Hello, world!
```

5. Other functions

5.1. Skip (`codly-skip`)

Convenience function for setting the skips, see the `skips` argument of the `codly` function.

5.2. Range (`codly-range`)

Convenience function for setting the range, see the `range` argument of the `codly` function. If you provide more than one range, as a list of arguments, it will set the `ranges` argument instead.

With a single range:

Example code Typst

```
#codly-range(2, end: 2)
```py
def fib(n):
  if n <= 1:
      return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```
  if n <= 1:
```

With more than one range:

Example code Typst

```
#codly-range(2, end: 2, (4, 5))
```py
def fib(n):
  if n <= 1:
      return n
  return fib(n - 1) + fib(n - 2)
fib(25)
```
```

Rendered output

```
  if n <= 1:
  return fib(n - 1) + fib(n - 2)
fib(25)
```

5.3. Offset (`codly-offset`)

Convenience function for setting the offset, see the `offset` argument of the `codly` function.

5.4. Local (`local`)

Codly provides a convenience function called `local` that allows you to locally override the global settings for a specific code block. This is useful when you want to apply a specific style to a code block without affecting the rest of the code blocks in your document. It works by overriding the default codly show rule locally with an override of the arguments by those you provide. It does not rely on states (much) and should no longer add layout passes to the rendering which could cause documents to not converge.

> ⚠ **Warning**
>
> When using `nested: false` on your local states, the outermost local state will be overriden by the inner local state(s). This means that the inner local state will be the only one that is applied to the code block. And that any previous local states (in the same hierarchy) will be ignored for subsequent code blocks.

> *i* **Info**
>
> Once custom elements become available in Typst, and codly moves to using those and set rules, this limitation will be lifted and you will be able to use nested local states without performance impact.

Example code Typst

````
*Global state with red color*
#codly(fill: red)
```typ
= Example
*Hello, World!*
```
*Locally set it to gray*
#local(
  fill: luma(240),
  ```typ
  = Example
  *Hello, World!*
  ```
)
* It's back to being red*
```typ
= Example
*Hello, World!*
```
````

Rendered output

Global state with red color

```typ
= Example
*Hello, World!*
```

Locally set it to gray

```typ
= Example
*Hello, World!*
```

It's back to being red

```typ
= Example
*Hello, World!*
```

5.4.1. Local state for per-language configuration

Additionally, local settings can be used to set per-language configuration using a show rule on your `raw` blocks. This can be done in one of two ways: by using a show rule on `raw.where(block: true, lang: "<lang>")` and calling the `local` function, or by using the `codly` function. The main differentiators are that the `local` function is faster and does not rely on states, while the `codly` function is more flexible, but slower and **will also style all following blocks**, you must therefore manually reset the changes.

> **Experiment**
>
> This should work in most cases, but this feature should be considered experimental. Please report any issues you encounter on GitHub: https://github.com/Dherse/codly.

> **Info**
>
> Note that you only want to do show rules on `raw` blocks where `block: true`, otherwise this will make your document slow.

> **Warning**
>
> If you use the `local` function in a show rule, nested `local` states **will not work** with the settings you have set! Use the `codly` method instead. If using the `codly` method, and you **must** manually reset the changed settings in the show rule!

Example code Typst

```
#show raw.where(block: true, lang: "rust"): local.with(
  number-format: numbering.with("I")
)

#show raw.where(block: true, lang: "py"): it => {
  codly(number-format: numbering.with("①"))
  it
  codly(number-format: numbering.with("1"))
}

*Numbered with Roman numerals*
```rust
fn main() {
  println!("Rust code has Roman numbers");
}
```

Numbered with circled numbers
```py
print("Python code has circled numbers")
```

Override with local state
#local(
  fill: blue.lighten(80%),
  ```py
  print("Python code is green")
  ```
)
```

Rendered output

Numbered with Roman numerals

```
I   fn main() {                                  rust
II    println!("Rust code has Roman numbers");
III }
IV
```

Numbered with circled numbers

```
①  print("Python code has circled numbers")      py
```

Override with local state

```
1  print("Python code is green")                  py
```

5.4.2. Nested local state

Codly does support nested local state, the innermost local state will override the outermost local state. This allows you to have different styles for different parts of your code block. This function takes the same arguments as the `codly` function, but only the arguments that are different from the global settings need to be provided.

> **⚠ Warning**
>
> Nested local states can slow down documents significantly if over-used (explicitly set `nested: true`). Use them sparingly and only when necessary. Another solution is to use the normal `codly` function before and after your code block. You can also use the the argument `nested: false` on `local` to prevent nested local states, which significantly reduces the performance impact.

Example code Typst

````
*Global state with red color*
#codly(fill: red)
```typ
= Example
*Hello, World!*
```
*Locally set it to blue*
#local(
  nested: true,
  fill: blue,
)[
  ```typ
  = Example
  *Hello, World!*
  ```
  *Now it's green:*
  #local(nested: true, fill: green)[
    ```typ
    = Example
    *Hello, World!*
    ```
  ]
  *Now its zebras are also blue:*
  #local(nested: true, zebra-fill: blue)[
    ```typ
    = Example
    *Hello, World!*
    ```
  ]

  *Back to blue:*
  ```typ
  = Example
  *Hello, World!*
  ```
]
*Back to red:*
```typ
= Example
*Hello, World!*
```
````

Rendered output

Global state with red color

```
1 = Example                typ
2 *Hello, World!*
```

Locally set it to blue

```
1 = Example                typ
2 *Hello, World!*
```

Now it's green:

```
1 = Example                typ
2 *Hello, World!*
```

Now its zebras are also blue:

```
1 = Example                typ
2 *Hello, World!*
```

Back to blue:

```
1 = Example                typ
2 *Hello, World!*
```

Back to red:

```
1 = Example                typ
2 *Hello, World!*
```

5.5. No codly (`no-codly`)

This is a convenience function equivalent to `local(enabled: false, body)`.

Example code Typst

```
*Enabled codly*
```typ
= Example
*Hello, World!*
```

Disabled codly
#no-codly[
  ```typ
  = Example
  *Hello, World!*
  ```
]
```

Rendered output

Enabled codly

```typ
= Example
*Hello, World!*
```

Disabled codly

= Example
Hello, World!

5.6. Enable (`codly-enable`)

Enables codly globally, equivalent to `codly(enabled: true)`.

Example code Typst

```
*Disabled codly*
#codly-disable()
```typ
= Example
*Hello, World!*
```
#codly-enable()
Enabled codly
```typ
= Example
*Hello, World!*
```
```

Rendered output

Disabled codly

= Example
Hello, World!

Enabled codly

```typ
= Example
*Hello, World!*
```

5.7. Disable (`codly-disable`)

Disables codly globally, equivalent to `codly(enabled: false)`.

Example code Typst

```
*Enabled codly*
```typ
= Example
*Hello, World!*
```

Disabled codly
#codly-disable()
```typ
= Example
*Hello, World!*
```
```

Rendered output

Enabled codly

```typ
= Example
*Hello, World!*
```

Disabled codly

= Example
Hello, World!

5.8. Reset (`codly-reset`)

Resets all codly settings to their default values. This is useful when you want to reset the settings of a code block to the default values after applying local settings.

Example code Typst

```typst
*Global state with red color*
#codly(fill: red)
```typ
= Example
*Hello, World!*
```
Reset it
#codly-reset()
```typ
= Example
*Hello, World!*
```
```

Rendered output

Global state with red color

```typ
= Example
*Hello, World!*
```

Reset it

```typ
= Example
*Hello, World!*
```

6. Codly performance