The codelst Package

A **Typst** package to render source code

v2.0.1 2023-07-19

Jonas Neugebauer

https://github.com/jneug/typst-codelst

codelst is a **Typst** package inspired by LaTeX packages like listings. It adds functionality to render source code with line numbers, highlighted lines and more.

Table of contents

Part I.
About

This package was created to render source code on my exercise sheets for my computer science classes. The exercises required source code to be set with line numbers that could be referenced from other parts of the document, to highlight certain lines and to load code from external files into my documents.

Since I used LaTeX before, I got inspired by packages like LISTINGS[1] and attempted to replicate some of its functionality. CODELST is the result of this effort.

This document is a full description of all available commands and options. The first part provides examples of the major features. The second part is a command reference for CODELST.

See `example.typ/example.pdf` for some quick examples how to use CODELST.

[1] https://ctan.org/package/listings

Part II.
Usage

II.1. Use as a package (Typst 0.9.0 and later)

For Typst 0.9.0 and later, CODELST can be imported from the preview repository:

```
#import "@preview/codelst:2.0.1": sourcecode
```

Alternatively, the package can be downloaded and saved into the system dependent local package repository.

Either download the current release from GitHub[2] and unpack the archive into your system dependent local repository folder[3] or clone it directly:

```
git clone https://github.com/jneug/typst-codelst.git codelst-2.0.1
```

In either case, make sure the files are placed in a folder with the correct version number: `codelst-2.0.1`

After installing the package, just import it inside your `typ` file:

```
#import "@local/codelst:2.0.1": sourcecode
```

II.2. Use as a module

To use CODELST as a module for one project, get the file `codelst.typ` from the repository and save it in your project folder.

Import the module as usual:

```
#import "codelst.typ": sourcecode
```

II.3. Rendering source code

CODELST adds the `#sourcecode()` command with various options to render code blocks. It wraps around any `#raw()` block to adds some functionality and formatting options to it:

[2] https://github.com/jneug/typst-codelst/releases/latest
[3] https://github.com/typst/packages#local-packages

````
#sourcecode[```typ
  #show "ArtosFlow": name => box[
    #box(image(
      "logo.svg",
      height: 0.7em,
    ))
    #name
  ]

  This report is embedded in the
  ArtosFlow project. ArtosFlow is a
  project of the Artos Institute.
```]
````

```typ
#show "ArtosFlow": name => box[
  #box(image(
    "logo.svg",
    height: 0.7em,
  ))
  #name
]

This report is embedded in the
ArtosFlow project. ArtosFlow is a
project of the Artos Institute.
```

CODELST adds line numbers and some default formatting to the code. Line numbers can be configured with a variety of options and ⟨frame⟩ sets a custom wrapper function for the code. Setting ⟨frame⟩: none disables the code frame.
```

```
    #sourcecode(
        numbers-side: right,
        numbering: "I",
        numbers-start: 10,
        numbers-first: 11,
        numbers-step: 4,
        numbers-style: (i) => align(right, text(fill:blue, emph(i))),
        frame: none
    )[```typ
    #show "ArtosFlow": name => box[
      #box(image(
        "logo.svg",
        height: 0.7em,
      ))
      #name
    ]

    This report is embedded in the
    ArtosFlow project. ArtosFlow is a
    project of the Artos Institute.
    ```]
```

---

```
#show "ArtosFlow": name => box[                                          XI
  #box(image(
    "logo.svg",
    height: 0.7em,
  ))
  #name                                                                  XV
]
This report is embedded in the
ArtosFlow project. ArtosFlow is a                                        XIX
project of the Artos Institute.
```

Since it is common to highlight code blocks by putting them inside a `#block()` element, CODELST does so with a light gray background and a border.

The frame can be modified by setting ⟨`frame`⟩ to a function with one argument. To do this globally, an alias for the `#sourcecode()` command can be created:
```

```
#let codelst-sourcecode = sourcecode
#let sourcecode = codelst-sourcecode.with(
  frame: block.with(
    fill: fuchsia.lighten(96%),
    stroke: 1pt + fuchsia,
    radius: 2pt,
    inset: (x: 10pt, y: 5pt)
  )
)

#sourcecode[```typ
#show "ArtosFlow": name => box[
  #box(image(
    "logo.svg",
    height: 0.7em,
  ))
  #name
]

This report is embedded in the
ArtosFlow project. ArtosFlow is a
project of the Artos Institute.
```]
```

---

```
1  #show "ArtosFlow": name => box[
2    #box(image(
3      "logo.svg",
4      height: 0.7em,
5    ))
6    #name
7  ]
8
9  This report is embedded in the
10 ArtosFlow project. ArtosFlow is a
11 project of the Artos Institute.
```

Line numbers can be formatted with the `numbers-style` option:
```

````
#sourcecode(
  gutter:2em,
    numbers-style:  (lno)  =>  text(fill:luma(120),  size:10pt,  emph(lno)  +
sym.arrow.r)
)[```typ
#show "ArtosFlow": name => box[
  #box(image(
    "logo.svg",
    height: 0.7em,
  ))
  #name
]

This report is embedded in the
ArtosFlow project. ArtosFlow is a
project of the Artos Institute.
```]
````

```
 1→   #show "ArtosFlow": name => box[
 2→     #box(image(
 3→       "logo.svg",
 4→       height: 0.7em,
 5→     ))
 6→     #name
 7→   ]
 8→
 9→   This report is embedded in the
10→   ArtosFlow project. ArtosFlow is a
11→   project of the Artos Institute.
```

CODELST handles whitespace in the code to save space and display the code as intended (and indented). Unnecessary blank lines at the beginning and end will be removed, alongside superfluous indentation:

````
#sourcecode[```java
    class HelloWorld {
      public static void main( String[] args ) {
        System.out.println("Hello World!");
      }
    }

```]
````

```
1  class HelloWorld {
2    public static void main( String[] args ) {
3      System.out.println("Hello World!");
4    }
5  }
```

This behavior can be disabled or modified:

```
#sourcecode(showlines:true, gobble:1, tab-size:4)[```java

class HelloWorld {
public static void main( String[] args ) {
System.out.println("Hello World!");
}
}

```]
```

---

```
1
2   class HelloWorld {
3     public static void main( String[] args ) {
4       System.out.println("Hello World!");
5     }
6   }
7
8
```

To show code from a file, load it with `#read()` and pass the result to `#sourcefile()` alongside the filename:

```
#sourcefile(read("typst.toml"), file:"typst.toml")
```

```
1  [package]
2  name = "codelst"
3  version = "2.0.1"
4  entrypoint = "codelst.typ"
5  authors = ["Jonas Neugebauer"]
6  license = "MIT"
7  description = "A typst package to render sourcecode."
8  repository = "https://github.com/jneug/typst-codelst"
9  exclude = ["example.typ", "example.pdf", "manual.pdf", "manual.typ",
   "tbump.toml"]
10 compiler = "0.0.9"
11 keywords = ["sourcecode", "code", "syntax-highlighting", "raw", "line
   numbers"]
```

It is useful to define an alias for `#sourcefile()`:

```
let codelst-sourcefile = sourcefile
let sourcefile( filename, ..args ) = codelst-sourcefile(
  read(filename), file:filename, ..args
)
```

`#sourcefile()` takes the same arguments as `#sourcecode()`. For example, to limit the output to a range of lines:
```

```
#sourcefile(
  showrange: (2, 4),
  read("typst.toml"),
file:"typst.toml"
)
```

```
2 name = "codelst"
3 version = "2.0.1"
4 entrypoint = "codelst.typ"
```

Specific lines can be highlighted:

```
#sourcefile(
  highlighted: (2, 3, 4),
  read("typst.toml"),
file:"typst.toml"
)
```

```
1  [package]
2  name = "codelst"
3  version = "2.0.1"
4  entrypoint = "codelst.typ"
5  authors = ["Jonas Neugebauer"]
6  license = "MIT"
7  description = "A typst package to render sourcecode."
8  repository = "https://github.com/jneug/typst-codelst"
9  exclude = ["example.typ", "example.pdf", "manual.pdf", "manual.typ",
   "tbump.toml"]
10 compiler = "0.0.9"
11 keywords = ["sourcecode", "code", "syntax-highlighting", "raw", "line
   numbers"]
```

To reference a line from other parts of the document, CODELST looks for labels in the source code and makes them available to Typst. The regex to look for labels can be modified to be compatible with different source syntaxes:

```
#sourcefile(
  label-regex: regex("\"(codelst.typ)\""),
  highlight-labels: true,
  highlight-color: lime,
  read("typst.toml"),
  file:"typst.toml"
)

See #lineref(<codelst.typ>) for the _entrypoint_.
```

```
1  [package]
2  name = "codelst"
3  version = "2.0.1"
4  entrypoint =
5  authors = ["Jonas Neugebauer"]
6  license = "MIT"
7  description = "A typst package to render sourcecode."
8  repository = "https://github.com/jneug/typst-codelst"
9  exclude = ["example.typ", "example.pdf", "manual.pdf", "manual.typ",
   "tbump.toml"]
10 compiler = "0.0.9"
11 keywords = ["sourcecode", "code", "syntax-highlighting", "raw", "line
   numbers"]
```

See line 4 for the *entrypoint*. (Note how the label was removed from the sourcecode before highlighting.)

II.4. Formatting

`#sourcecode()` can be used inside `#figure()` and will show the correct supplement. It is recommended to allow page breaks for `raw` figures:

```
11 #show figure.where(kind: raw): set block(breakable: true)
```

Instead of the build in styles, custom functions can be used:

```
#sourcecode(
  numbers-style: (lno) => text(
    size: 2em,
    fill:rgb(220, 65, 241),
    font:("Comic Sans MS"),
    str(lno)
  ),
  frame: (code) => block(
    width:100%,
    inset:(x:10%, y:0pt),
    block(fill: green, width:100%, code)
), raw("*some*
_source_
= code", lang:"typc"))
```

```
1 *some*
2 _source_
3 = code
```

Using other packages like SHOWYBOX is easy:

```
#import "@preview/showybox:2.0.1": showybox

#let showycode = sourcecode.with(
  frame: (code) => showybox(
    frame: (
      title-color: red.darken(40%),
      body-color: red.lighten(90%),
      border-color: black,
      thickness: 2pt
    ),
    title: "Source code",
    code
  )
)

#showycode[```typ
*some*
_source_
= code
```]
```

---

Source code

```
1 *some*
2 _source_
3 = code
```

This is nice in combination with figures:
```

```
#import "@preview/showybox:2.0.1": showybox

#show figure.where(kind: raw): (fig) => showybox(
  frame: (
    title-color: red.darken(40%),
    body-color: red.lighten(90%),
    border-color: black,
    thickness: 2pt
  ),
  title: [#fig.caption.body #h(1fr) #fig.supplement #fig.counter.display()],
  fig.body
)

#figure(
  sourcecode(frame: none)[```typ
    *some*
    _source_
    = code
  ```],
  caption: "Some code"
)
```

---

Some code — Listing 1

```
1 *some*
2 _source_
3 = code
```

II.4.1. Using codelst for all raw text

> Since Typst 0.9.0 using a `#show` rule should become possible, but not yet fully implemented in CODELST.

Using a `#show` rule to set all `#raw()` blocks inside `#sourcecode()` is not possible, since the command internally creates a new `#raw()` block and would cause Typst to crash with an overflow error. Using a custom ⟨lang⟩ can work around this, though:

```
#show raw.where(lang: "clst-typ"): (code) => sourcecode(lang:"typ", code)

```clst-typ
*some*
_source_
= code
```
```

---

```
1 *some*
2 _source_
3 = code
```

CODELST provides two ways to get around this issue, however. One is to set up a custom language that is directly followed by a colon and the true language tag:
```

```
:typ
*some*
_source_
= code
```

This is a robust way to send anything to CODELST. But since this might prevent proper syntax highlighting in IDEs, a reversed syntax is possible:

```
:codelst
*some*
_source_
= code
```

This will look at the first line of every `raw` text and if it matches `:codelst`, it will remove the activation tag and send the code to `#sourcecode()`.

Setting up one of these catchall methods is easily done by using the `#codelst()` function in a `#show` rule. Any arguments will be passed on to `#sourcecode()`:

```
#show: codelst( ..sourcecode-args )

// or

#show: codelst( reversed: true, ..sourcecode-args )
```

II.5. Command overview

```
#sourcecode(
  ⟨lang⟩: auto,
  ⟨numbering⟩: "1",
  ⟨numbers-start⟩: auto,
  ⟨numbers-side⟩: left,
  ⟨numbers-width⟩: auto,
  ⟨numbers-style⟩: "function",
  ⟨numbers-first⟩: 1,
  ⟨numbers-step⟩: 1,
  ⟨gutter⟩: 10pt,
  ⟨tab-indent⟩: 2,
  ⟨gobble⟩: auto,
  ⟨highlighted⟩: (),
  ⟨highlight-color⟩: rgb("#eaeabd"),
  ⟨label-regex⟩: regex("// <([a-z-]{3,})>$"),
  ⟨highlight-labels⟩: false,
  ⟨showrange⟩: none,
  ⟨showlines⟩: false,
  ⟨frame⟩: "code-frame"
)[⟨code⟩]
```

Argument

⟨numbering⟩: "1" string | function | none

A numbering pattern to use for line numbers. Set to none to disable line numbers.

Argument

⟨numbers-start⟩: auto integer | auto

The number of the first code line. If set to auto, the first line will be set to the start of ⟨showrange⟩ or 1 otherwise.

Argument

⟨numbers-side⟩: left|right alignment

On which side of the code the line numbers should appear.

Argument

⟨numbers-width⟩: auto auto | length

The width of the line numbers column. Setting this to auto will measure the maximum size of the line numbers and size the column accordingly. Giving a negative length will move the numbers out of the frame into the margin.

Argument

⟨numbers-first⟩: 1 none

The first line number to show. Compared to ⟨numbers-start⟩, this will not change the numbers but hide all numbers before the given number.

Argument

⟨numbers-step⟩: 1 none

The step size for line numbers. For ⟨numbers-step⟩: n only every n-th line number is shown.

Argument

⟨numbers-style⟩: (i) => i function

A function of one argument to format the line numbers. Should return content.

Argument

⟨gutter⟩: 10pt none

Gutter between line numbers and code lines.

Argument

⟨tab-indent⟩: 2 none

Number of spaces to replace tabs at the start of each line with.

Argument

⟨gobble⟩: auto auto | integer | boolean

How many whitespace characters to remove from each line. By default, the number is automatically determined by finding the maximum number of whitespace all lines have in common. If ⟨gobble⟩: false, no whitespace is removed.

Argument

⟨highlighted⟩: () — none

Line numbers to highlight.

Note that the numbers will respect ⟨numbers-start⟩. To highlight the second line with ⟨numbers-start⟩: 15, pass ⟨highlighted⟩: (17,)

Argument

⟨highlight-color⟩: rgb("#eaeabd") — none

Color for highlighting lines.

Argument

⟨label-regex⟩ — regular expression

A regular expression for matching labels in the source code. The default value will match labels with at least three characters at the end of lines, separated with a line comment (//). For example:

```
#strong[Some text] // <my-line-label>
```

If this line matches on a line, the full match will be removed from the output and the content of the first capture group will be used as the label's name (my-line-label in the example above).

Note that to be valid, the expression needs to have at least one capture group.

To reference a line, #lineref() should be used.

Argument

⟨highlight-labels⟩: false — none

If set to true, lines matching ⟨label-regex⟩ will be highlighted.

Argument

⟨showrange⟩: none — none | array

If set to an array with exactly two integers, the code-lines will be sliced to show only the lines within that range.

For example, ⟨showrange⟩: (5, 10) will only show the lines 5 to 10.

If settings this and ⟨numbers-start⟩: auto, the line numbers will start at the number indicated by the first number in ⟨showrange⟩. Otherwise, the numbering will start as specified with ⟨numbers-start⟩.

Argument

⟨showlines⟩: false — none

If set to `true`, no blank lines will be stripped from the start and end of the code. Otherwise, those lines will be removed from the output.

Line numbering will not be adjusted to the removed lines (other than with ⟨showrange⟩).

Argument

⟨frame⟩: "code-frame" `function`

A function of one argument to frame the source code. The default is `#code-frame()`. `none` disables any frame.

#sourcefile(⟨code⟩, ⟨file⟩: none, ⟨lang⟩: auto, ..⟨args⟩)

Takes a text string ⟨code⟩ loaded via the `#read()` function and passes it to `#sourcecode()` for display. If ⟨file⟩ is given, the code language is guessed by the file's extension. Otherwise, ⟨lang⟩ can be provided explicitly.

Any other ⟨args⟩ will be passed to `#sourcecode()`.

```
#sourcefile(read("typst.toml"), file:"typst.toml")
```

```
[package]
name = "codelst"
version = "2.0.1"
entrypoint = "codelst.typ"
authors = ["Jonas Neugebauer"]
license = "MIT"
description = "A typst package to render sourcecode."
repository = "https://github.com/jneug/typst-codelst"
exclude = ["example.typ", "example.pdf", "manual.pdf", "manual.typ", "tbump.toml"]
compiler = "0.0.9"
keywords = ["sourcecode", "code", "syntax-highlighting", "raw", "line numbers"]
```

The idea for `#sourcefile()` was to read the provided filename without the need for the user to call `#read()`. Due to the security measure, that packages can only read files from their own directory, the call to `#read()` needs to happen outside `#sourcefile()` in the document.

For this reason, the command differs from `#sourcecode()` only insofar as it accepts a `string` instead of `raw content`.

Future releases might use the ⟨filename⟩ for other purposes, though.

To deal with this, simply add the following code to the top of your document to define a local alias for `#sourcefile()`:

```
#let codelst-sourcefile = sourcefile
#let sourcefile( filename, ..args ) = codelst-sourcefile(read(filename), file:filename, ..args)
```

#lineref(⟨label⟩, ⟨supplement⟩: "line")

Creates a reference to a code line with a label. ⟨label⟩ is the label to reference.

````
#sourcecode[```java
class HelloWorld {
  public static void main( String[] args ) {
    System.out.println("Hello World!");
  }
}
```]

See #lineref(<main-method>) for a main method in Java.
````

```java
class HelloWorld {
  public static void main( String[] args ) {
    System.out.println("Hello World!");
  }
}
```

See line 2 for a main method in Java.

How to set labels for lines, refer to the documentation of ⟨label-regex⟩ at #name().

#code-frame(⟨fill⟩: luma(98.04%), ⟨stroke⟩: 1pt + luma(78.43%), ⟨inset⟩: (x: 5pt, y: 10pt), ⟨radius⟩: 4pt)[⟨code⟩]

Convenience function to create a #block() to wrap code inside. The arguments are passed to block.

The default values create the default gray box around source code.

Should be used with the ⟨frame⟩ argument in #sourcecode().

````
#code-frame(lorem(20))
````

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

````
#sourcecode(
  frame: code-frame.with(
    fill:   green.lighten(90%),
    stroke: green
  )
)[```typc
lorem(20)
```]
````

```typc
lorem(20)
```

#codelst(⟨tag⟩: "codelst", ⟨reversed⟩: false, ..⟨sourcecode-args⟩)

Sets up a default style for raw blocks. Read Section II.4.1 for details on how it works.

```
#show: codelst()
```

Part III.
Limitations and alternatvies

III.1. Limitations and Issues

To lay out the code and line numbers correctly, CODELST needs to know the available space before calculating the correct sizes. This will lead to problems when changing the layout of the code later on, for example with a `#show` rule.

The way line numbers are laid out, the alignment might drift off for large code blocks. Page breaks are a major cause for this. If applicable, it can help to split large blocks of code into smaller chunks, for example by using ⟨`showrange`⟩.

The insets for line highlights are slightly off.

III.2. Alternatives

There are some alternatives to CODELST that fill similar purposes, but have more or other functionality. If CODELST does not suit your needs, one of those might do the trick.

platformer/typst-algorithms[4] *Typst module for writing algorithms. Use the algo function for writing pseudocode and the code function for writing code blocks with line numbers.*

hugo-s29/typst-algo[5] *This package helps you typeset [pseudo] algorithms in Typst.*

[4] https://github.com/platformer/typst-algorithms
[5] https://github.com/hugo-s29/typst-algo

Part IV.
Index