code-link

github.com/EagleoutIce/xlistings

Florian Sihler*

Version v1.1.0 – 2026/11/11

code-link turns the code of your listings into hyperlinks. You register a string where it is defined, and every occurrence of this string in a listings, xlistings or minted block links to that place. It works for every language these packages can typeset, and for the text between the listings as well.

```
\CodeLinkAnchor[group=java]{Greeter}
The class \CodeLinkRef[group=java]{Greeter} greets:
\begin{lstlisting}[language=java]
Greeter g = new Greeter("world");
g.greet(); // a Greeter, again
\end{lstlisting}
```

The class Greeter greets:

```
Greeter g = new Greeter("world");
g.greet(); // a Greeter, again
```

The package is written in expl3 and needs only l3keys2e. Links are made by hyperref, by any package that provides \hyperref, or by a command of your own (Section 8.3). Links come from the .aux file, like references, so they need no run beyond the second one that hyperref needs anyway. With 1 000 strings, a run takes 128 ms longer, and each listing of 500 tokens another 9 ms: about 19 % of a run of the benchmark document, where a further run would cost 100 % (Section 10).

A key or command with a badge works only with the backends named by it. Without a badge it works with all of them. The badges listings and xlistings mark keys that only these two understand, and minted marks keys that only minted understands.

1 Quick start 1
2 Commands 2
3 Options 2
4 Groups and styles 3
5 Languages 3
6 What is matched 5
7 Looks of the links 6
8 Backends and link providers 8
9 Package options 9
10 Benchmark 10
11 Limitations 10
Index of commands, keys and names 11

code-link is a package of its own, developed in the same repository as xlistings because the two are most useful combined. It does not need xlistings and works with listings or minted alone. The manual of xlistings is a document of its own. The package was extracted from tikzpingus.

*Released under the LaTeX Project Public License, version 1.3c or later.

1 Quick start

Load a listings backend, `hyperref` and the package. Mark the place where something is defined with `\CodeLinkAnchor`, and every listing and every `\CodeLinkRef` that contains the string links there.

```
\usepackage{listings}   % or xlistings, or minted
\usepackage{hyperref}   % or any package which provides \hyperref
\usepackage[underline]{code-link}
...
\CodeLinkAnchor[command]{mymacro}   % defines the target
\CodeLinkRef[command]{mymacro}      % links in the text
```

Strings are stored in the `.aux` file while the document is typeset. Like references, a link appears from the second run on, and it also points forward.

```
\CodeLinkAnchor[group=py]{fibonacci}
Calling \CodeLinkRef[group=py]{fibonacci}:
\begin{lstlisting}[language=python]
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)
\end{lstlisting}
```

Calling fibonacci:

```python
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Without the key **language**=⟨*name*⟩ (Section 4), a string is linked in the listings of every language.

2 Commands

All of these take [⟨*options*⟩]: the keys of Section 3, for example **group**=⟨*name*⟩ or **command**.

- `\CodeLinkAnchor`[⟨*options*⟩]{⟨*string*⟩} places a target here (a `hyperref` anchor and a `\label`) and registers the string. Only the first registration of a string counts per language.
- `\CodeLinkRegister`[⟨*options*⟩]{⟨*string*⟩} registers the string for a target that exists elsewhere, for example a heading or a theorem. Give the target with **label**=⟨*label*⟩.
- `\CodeLinkBlock`[⟨*options*⟩]{⟨*string*⟩} registers a string that is no link but is matched as one unit. The `out` inside `System.out` is not linked here, although `out` is registered:

  ```
  \CodeLinkAnchor[group=java]{out}
  \CodeLinkBlock[group=java]{System.out}
  \begin{lstlisting}[language=java]
  System.out.println(out); // only the last out
  \end{lstlisting}
  ```

  ```java
  System.out.println(out); // only the last out
  ```

- `\CodeLinkRef`[⟨*options*⟩]{⟨*string*⟩} typesets the string as a link in running text, in the font `\CodeLinkTextFont` (default `\ttfamily`).
- `\CodeLinkLabel`[⟨*group*⟩]{⟨*string*⟩} expands to the label of a string, so `\ref{\CodeLinkLabel{foo}}` works. The star form is for commands. A label is the prefix of the group and the string, where special characters are replaced (`\` by `:bs:`, space by `-`):

  ```
  \CodeLinkAnchor[group=py]{answer}
  Page \pageref{\CodeLinkLabel[py]{answer}},
  label \T{\CodeLinkLabel[py]{answer}}
  ```

  Page 2, label `py:answer`

3 Options

These are the options of `\CodeLinkAnchor`, `\CodeLinkRegister`, `\CodeLinkBlock` and `\CodeLinkRef`. A group (Section 4) sets the defaults of **`prefix`**=⟨*text*⟩, **`style`**=⟨*name*⟩, **`language`**=⟨*name*⟩, **`links`**=`true|false`, **`highlight`**=`true|false` and **`boundary`**=`true|false`.

- **`group`**=⟨*name*⟩ (default `default`) the group of the string.
- **`command`** (default `off`) the string is a control sequence. The backslash is added, so `[command]{node}` registers `\node`.
- **`label`**=⟨*label*⟩ (default `generated`) the label of the target.
- **`display`**=⟨*text*⟩ (default `the string`) listings xlistings the text shown instead of the string. It is typeset literally.
- **`style`**=⟨*name*⟩ (default `of the group`) a style defined with `\CodeLinkStyle`.
- **`language`**=⟨*name*⟩ (default `of the group`) link only in listings of this language (listings: the key `language`, minted: the lexer). Empty means all.
- **`links`**=`true|false` (default `of the group`) `false`: the string is registered but never linked. It stays plain text.
- **`highlight`**=`true|false` (default `of the group, false`) `false`: a listing keeps its syntax highlighting, so the style of the string is used in the text only and the underline marks the link. `true`: the style is used in listings as well (Section 7.1).
- **`boundary`**=`true|false` (default `of the group`) listings xlistings `false`: a string that is no word may also be found at the start of a longer word (Section 6).

4 Groups and styles

- `\CodeLinkGroup{`⟨*name*⟩`}{`⟨*keys*⟩`}` defines the defaults of a group. The keys are **`prefix`**=⟨*text*⟩ (the start of the labels, default `cl:`), **`style`**=⟨*name*⟩ (default `none`), **`language`**=⟨*name*⟩ (default: all languages) and the three switches **`links`**, **`highlight`** (default `false`) and **`boundary`** of Section 3. Names of languages are compared in lowercase and without the dialect.
- `\CodeLinkStyle{`⟨*name*⟩`}{`⟨*code*⟩`}` defines a style. Its code runs inside of the group that typesets the link, for example `\bfseries\color{red}`. The style `none` does nothing, so by default a link changes neither color nor font.

The groups of this document:

```
\CodeLinkStyle{def}{\bfseries}
\CodeLinkStyle{api}{\color{clcmd}}
\CodeLinkGroup{api}{prefix=api:,style=api}
\CodeLinkGroup{java}{prefix=java:,language=java}
\CodeLinkGroup{py}{prefix=py:,language=python}
\CodeLinkGroup{hs}{prefix=hs:,language=haskell}
```

5 Languages

The strings do not have to be LaTeX commands. Whatever a listing language knows as a name can be linked. The groups scope their strings to a language, so the same name may be defined in each language, and every listing links to the definition in its own language. We define a stack in Java, Python and Haskell. All three have a `push` and a `size`.

5.1 Java

The class `Stack` has the methods `push` and `size`. The anchors are written at the start of the example:

```
\CodeLinkAnchor[group=java]{Stack}\CodeLinkAnchor[group=java]{push}\CodeLinkAnchor[group=java]{size}
\begin{lstlisting}[language=java]
public class Stack {
    private final Deque<Integer> items = new ArrayDeque<>();
    public void push(int x) { items.addFirst(x); }
    public int size() { return items.size(); }
}
\end{lstlisting}
```

```java
public class Stack {
    private final Deque<Integer> items = new ArrayDeque<>();
    public void push(int x) { items.addFirst(x); }
    public int size() { return items.size(); }
}
```

The `size` of `items` is linked as well, because it is the same name in the same language.

5.2 Python

The Python class has the same names, but its definitions are different targets:

```
\CodeLinkAnchor[group=py]{Stack}\CodeLinkAnchor[group=py]{push}\CodeLinkAnchor[group=py]{size}
\begin{lstlisting}[language=python]
class Stack:
    def __init__(self): self.items = []
    def push(self, x): self.items.append(x)
    def size(self): return len(self.items)
\end{lstlisting}
```

```python
class Stack:
    def __init__(self): self.items = []
    def push(self, x): self.items.append(x)
    def size(self): return len(self.items)
```

Click through the Java and the Python listing. The push of one never jumps to the other.

5.3 Haskell

In Haskell, the functions are defined by their type signature. The text may talk about all three languages at once, because `\CodeLinkRef` and `\lstinline` know the language as well. `stack.push(1)`, `stack.push(1)` and `push 1 stack` link to three places.

```
\CodeLinkAnchor[group=hs]{Stack}\CodeLinkAnchor[group=hs]{push}\CodeLinkAnchor[group=hs]{size}
\begin{lstlisting}[language=haskell]
newtype Stack a = Stack [a]

push :: a -> Stack a -> Stack a
push x (Stack xs) = Stack (x : xs)

size :: Stack a -> Int
size (Stack xs) = length xs
\end{lstlisting}
```

```haskell
newtype Stack a = Stack [a]

push :: a -> Stack a -> Stack a
push x (Stack xs) = Stack (x : xs)

size :: Stack a -> Int
size (Stack xs) = length xs
```

5.4 A reference with an index

\CodeLinkAnchorHook{⟨*code*⟩} runs at every anchor, with the string as `#1` and the label as `#2`. The definitions of an API reference can fill the index with it:

```
\CodeLinkAnchorHook{\index{#1@\texttt{#1} (Python)\string|hyperpage}}
\CodeLinkAnchor[group=py]{Counter}\CodeLinkAnchor[group=py]{update}\CodeLinkAnchor[group=py]{elements}

\noindent\textbf{\CodeLinkRef[group=py]{Counter}} counts hashable objects. It knows \CodeLinkRef[group=py]{update} (add
↪counts) and \CodeLinkRef[group=py]{elements} (every item as often as it was counted):
\begin{lstlisting}[language=python]
from collections import Counter

words = Counter("a b a c b a".split())
words.update(["c", "c"])
print(sorted(words.elements()))   # a, a, a, b, b, c, c, c
\end{lstlisting}
\CodeLinkAnchorHook{}
```

Counter counts hashable objects. It knows update (add counts) and elements (every item as often as it was counted):

```python
from collections import Counter

words = Counter("a b a c b a".split())
words.update(["c", "c"])
print(sorted(words.elements()))   # a, a, a, b, b, c, c, c
```

Counter, update and elements are now in the index of this document.

6 What is matched

A word is a name made of letters, or a command (a backslash and letters). Words are matched as a whole, so drop is not linked inside of dropped. Everything else is handed to the listings backend as a rule (Section 8).

A word. It is linked where it stands alone:

```
\CodeLinkAnchor[group=java]{Greeting}
\begin{lstlisting}[language=java]
Greeting a; GreetingCard b;
\end{lstlisting}
```

```java
Greeting a; GreetingCard b;
```

A command. The key **command** adds the backslash. A command is a word as well, so `\textbfx` is no `\textbf`:

```
\CodeLinkAnchor[group=tex,command]{textbf}
\begin{lstlisting}[language={[LaTeX]TeX}]
\textbf{bold} and \textbfx
\end{lstlisting}
```

```
\textbf{bold} and \textbfx
```

A name with an underscore. It is no word, so it is matched as a rule. A rule would also fire at the start of a longer name. The key **boundary**=true (the default) prevents this:

```
\CodeLinkAnchor[group=py]{max_size}
\begin{lstlisting}[language=python]
max_size = 3; max_size_x = 4
\end{lstlisting}
```

```
max_size = 3; max_size_x = 4
```

Without the boundary. With **boundary**=false, the rule fires at the start of the longer name as well. This costs less time, because no additional rules are made:

```
\CodeLinkAnchor[group=py,boundary=false]{min_size}
\begin{lstlisting}[language=python]
min_size = 3; min_size_x = 4
\end{lstlisting}
```

```
min_size = 3; min_size_x = 4
```

A name with digits. Digits are numbers to `xlistings`, so `utf8` is a rule as well:

```
\CodeLinkAnchor[group=py]{utf8}
\begin{lstlisting}[language=python]
open(path, encoding="utf8")
\end{lstlisting}
```

```
open(path, encoding="utf8")
```

Dots and primes. Qualified names and names with a prime are rules:

```
\CodeLinkAnchor[group=hs]{Data.Map}
\CodeLinkAnchor[group=hs]{foldr'}
\begin{lstlisting}[language=haskell]
import qualified Data.Map as M
total = foldr' (+) 0 (M.elems m)
\end{lstlisting}
```

```
import qualified Data.Map as M
total = foldr' (+) 0 (M.elems m)
```

Operators. Symbols are rules as well:

```
\CodeLinkAnchor[group=hs]{<$>}
\begin{lstlisting}[language=haskell]
names = show <$> ids
\end{lstlisting}
```

```
names = show <$> ids
```

Several words. listings xlistings A string may contain spaces. Pygments works on single tokens, so `minted` never links such a string (Section 8.2):

```
\CodeLinkAnchor[group=java]{new Stack}
\begin{lstlisting}[language=java]
var s = new Stack(); // new Stack
\end{lstlisting}
```

```
var s = new Stack(); // new Stack
```

7 Looks of the links

A link is typeset by \CodeLinkFormat{⟨*label*⟩}{⟨*style*⟩}{⟨*text*⟩}. The style of the group is applied first, and the default format then makes the text a link. The package option **underline** (default: off) draws a slight line below every link. The line is thin and light, and it lies at a fixed distance below the baseline, no matter how deep the word is. The links of a line share it:

```
\CodeLinkAnchor[group=py]{quartz}
\CodeLinkAnchor[group=py]{gypsy}
\CodeLinkAnchor[group=py]{ill}
{\Large\CodeLinkRef[group=py]{ill}
\CodeLinkRef[group=py]{gypsy}
\CodeLinkRef[group=py]{quartz}}
```

ill gypsy quartz

The keys **underline-offset**=⟨*length*⟩ (default 1.2 pt), **underline-thickness**=⟨*length*⟩ (0.3 pt) and **underline-color**=⟨*color*⟩ (black!15) change the line. The color is an xcolor expression. An empty color uses the color of the text, and without xcolor the line has that color as well. In running text the line has the color **underline-text-color**=⟨*color*⟩ (black!8).

```
\CodeLinkSetup{underline-offset=2pt, underline-thickness=.6pt, underline-color=Indigo!35}
```

7.1 Syntax highlighting

By default a link changes neither color nor font in a listing. The listing keeps its syntax highlighting, and the link adds the hyperlink and the underline. The style of a group applies in the text, for example in \CodeLinkRef. With **highlight**=true it applies in listings as well, and a color in it replaces the color of the highlighting for the linked words. Here, class is registered in a group with a red style and **highlight**=true:

```
\CodeLinkStyle{red}{\color{red}}
\CodeLinkGroup{hlred}{prefix=hlred:,language=java,style=red,highlight=true}
\CodeLinkAnchor[group=hlred]{class}
\begin{lstlisting}[language=java]
public class Stack {}
\end{lstlisting}
```

```
public class Stack {}
```

The same style without the key keeps the color of the highlighting. The keyword public below is linked, and its color is still the color of the keywords:

```
\CodeLinkGroup{hlkeep}{prefix=hlkeep:,language=java,style=red}
\CodeLinkAnchor[group=hlkeep]{public}
\begin{lstlisting}[language=java]
public class Stack {}
\end{lstlisting}
```

```java
public class Stack {}
```

7.2 No links

A group with **links**=false registers nothing, so its strings stay plain text. The key works for a single string as well:

```
\CodeLinkGroup{quiet}{prefix=quiet:,links=false}
\CodeLinkAnchor[group=quiet]{Queue}
\begin{lstlisting}[language=java]
Queue q = new Queue();   // not linked
\end{lstlisting}
```

```java
Queue q = new Queue();   // not linked
```

The listings key **codelink**=false listings xlistings switches the links of one listing off, even for registered strings:

```
\begin{lstlisting}[language=java,codelink=false]
Stack s = new Stack();   // not linked, the listing has codelink=false
\end{lstlisting}
```

```java
Stack s = new Stack();   // not linked, the listing has codelink=false
```

Without the key, the same listing is linked:

```
\begin{lstlisting}[language=java]
Stack s = new Stack();   // linked
\end{lstlisting}
```

```java
Stack s = new Stack();   // linked
```

7.3 Formats

Redefine \CodeLinkFormat to change what a link looks like. The format below adds the page of the target. You then get only what you wrote, without the underline:

```
\CodeLinkAnchor[group=py]{looks}
\begingroup
\RenewDocumentCommand\CodeLinkFormat{mmm}{\hyperref[#1]{#3}\textsuperscript{\scriptsize\pageref*{#1}}}
A format with the page: \CodeLinkRef[group=py]{looks}
\endgroup
```

A format with the page: looks[8]

8 Backends and link providers

8.1 listings and xlistings

Words are checked when `listings` outputs an identifier, which costs next to nothing for any number of strings. All other strings are given to `listings` as `literate` rules. They follow the setting `literate*` (no links in comments and strings if starred), they apply inside of `\lstinline`, and they may contain spaces. The longest string wins. A rule matches characters, so without a boundary it also fires inside of a longer word. `xlistings` needs no special treatment.

8.2 minted

Pygments splits the code into tokens, and `code-link` links a token if it is exactly equal to a registered string. Words work as in `listings`. Strings of several tokens are never linked, because no single token equals them. `\CodeLinkBlock` works across tokens: a token is not linked if the text before it ends with a block, so the `out` of `System.out` stays plain. The keys **`display`**=⟨*text*⟩ and **`boundary`** have no effect. `minted` version 3 and `-shell-escape` are needed.

```
\usepackage{minted}
\usepackage[underline]{code-link}
...
\CodeLinkAnchor[group=java]{Greeter}
\CodeLinkAnchor[group=java]{out}
\CodeLinkBlock[group=java]{System.out}
\begin{minted}{java}
System.out.println(new Greeter(out));
\end{minted}
```

The result is below. `Greeter` and the last `out` are links:

```java
System.out.println(new Greeter(out));
```

8.3 Link providers

A link is made by `\CodeLinkLinkCommand`{⟨*code*⟩}. In the code, **`#1`** is the label and **`#2`** the text. The default is `\hyperref[#1]{#2}`, so `hyperref` and every package that provides `\hyperref` work as they are. For other reference packages, or none, write the link yourself:

```
\CodeLinkAnchor[group=py]{target}
\begingroup
\CodeLinkLinkCommand{#2\textsuperscript{[\pageref{#1}]}}
A page reference as the link: \CodeLinkRef[group=py]{target}
\endgroup
```

A page reference as the link: `target`[9]

9 Package options

`\CodeLinkSetup`{⟨*keys*⟩} takes the options of the package as well:

- **`backend`**=`auto|listings|minted|none` (default `auto`) the backend to extend. By default it is whatever is loaded.
- **`hyper`**=`auto|true|false` (default `auto`) use links. `auto` means yes if `\hyperref` exists or a link command was set.
- **`underline`** (default `off`) draw a line below the links.

- **underline-offset**=⟨*length*⟩ (default 1.2 pt) the distance of the line from the baseline.
- **underline-thickness**=⟨*length*⟩ (default 0.3 pt) the stroke of the line.
- **underline-color**=⟨*color*⟩ (default `black!15`) the color of the line.
- **underline-text-color**=⟨*color*⟩ (default `black!8`) the color of the line below links in the text. It is lighter than in listings.
- **underline-text**=`true|false` (default `true`) `false`: links in the text have no line, only listings do.
- **debug** (default `off`) print what the package found.

10 Benchmark

`bench/run.sh` measures the cases of `bench/cases.txt` with the timer of pdfTEX. Every case runs twice, as the strings are read from the `.aux` file, and five times over; the tables show the average. A listing has 10 tokens per line, half of them registered.

- **Plain**, **Linked** one listing without and with links; **Extra** is the difference, paid per listing.
- **Register** all `\CodeLinkAnchor` calls, paid once per run.
- **Assemble** reading the strings and building the rules, paid once per run.
- **Overhead** Register, Assemble and Extra together, and in parentheses their share of **Run**.
- **Run** the wall time of the whole pdfLATEX run of the benchmark document, process start included. A document that needed a further run would pay this time again, 100 %.

With 1 000 strings, a run costs 128 ms more and each listing 9 ms more for words (159 ms and 15 ms for commands, 107 ms and 7 ms with `minted`). That is 19 % of a run with `listings` and 8 % with `minted`, whose runs are longer.

listings

Case	Strings	Tokens	Plain	Linked	Extra	Register	Assemble	Overhead	Run
no strings	0	500	4.5 ms	4.7 ms	0.2 ms	0.0 ms	0.0 ms	0.2 ms (0 %)	212.0 ms
10 words	10	500	4.7 ms	9.8 ms	5.1 ms	1.0 ms	0.5 ms	6.7 ms (3 %)	238.3 ms
100 words	100	500	4.6 ms	10.5 ms	5.8 ms	8.5 ms	4.4 ms	18.7 ms (8 %)	244.6 ms
1000 words	1000	500	4.6 ms	13.4 ms	8.8 ms	79.6 ms	43.3 ms	131.7 ms (20 %)	672.2 ms
1000 words, long listing	1000	2500	21.0 ms	49.7 ms	28.7 ms	79.7 ms	42.3 ms	150.7 ms (19 %)	805.4 ms
10 commands	10	500	8.0 ms	19.4 ms	11.4 ms	1.4 ms	0.5 ms	13.4 ms (4 %)	300.6 ms
100 commands	100	500	8.9 ms	20.0 ms	11.1 ms	12.1 ms	5.3 ms	28.5 ms (8 %)	364.2 ms
1000 commands	1000	500	9.2 ms	24.8 ms	15.6 ms	114.2 ms	49.4 ms	179.2 ms (20 %)	889.2 ms

xlistings

Case	Strings	Tokens	Plain	Linked	Extra	Register	Assemble	Overhead	Run
no strings	0	500	7.2 ms	7.4 ms	0.2 ms	0.0 ms	0.0 ms	0.2 ms (0 %)	292.3 ms
10 words	10	500	6.9 ms	12.3 ms	5.4 ms	1.2 ms	0.5 ms	7.0 ms (2 %)	327.8 ms
100 words	100	500	6.7 ms	15.3 ms	8.6 ms	8.9 ms	5.3 ms	22.9 ms (7 %)	345.0 ms
1000 words	1000	500	6.1 ms	14.4 ms	8.3 ms	88.4 ms	45.1 ms	141.8 ms (19 %)	742.9 ms
1000 words, long listing	1000	2500	24.2 ms	58.7 ms	34.6 ms	89.8 ms	49.5 ms	173.9 ms (18 %)	948.6 ms
10 commands	10	500	14.8 ms	28.6 ms	13.8 ms	2.4 ms	0.8 ms	17.1 ms (4 %)	463.0 ms
100 commands	100	500	9.0 ms	18.7 ms	9.7 ms	10.8 ms	4.7 ms	25.2 ms (7 %)	335.8 ms
1000 commands	1000	500	8.8 ms	23.1 ms	14.3 ms	104.6 ms	50.7 ms	169.6 ms (21 %)	819.2 ms

minted, with cached Pygments output

Case	Strings	Tokens	Plain	Linked	Extra	Register	Assemble	Overhead	Run
no strings	0	500	7.1 ms	7.3 ms	0.1 ms	0.0 ms	0.0 ms	0.1 ms (0 %)	1 085.4 ms
10 words	10	500	6.6 ms	9.5 ms	2.9 ms	1.1 ms	0.1 ms	4.1 ms (0 %)	853.7 ms
100 words	100	500	6.6 ms	11.0 ms	4.5 ms	8.6 ms	1.1 ms	14.1 ms (1 %)	945.6 ms
1000 words	1000	500	7.3 ms	14.4 ms	7.2 ms	95.7 ms	11.2 ms	114.1 ms (8 %)	1 446.9 ms
1000 words, long listing	1000	2500	22.0 ms	39.6 ms	17.7 ms	79.1 ms	9.2 ms	105.9 ms (8 %)	1 290.5 ms

The cost of a listing grows with its links, not with the number of strings, because words are looked up once per identifier. Rules are different. `listings` tries them one after another at every occurrence of their first character, so their cost grows with their number. If you have many strings, prefer words.

11 Limitations

Strings are written to the main `.aux` file. With pdfTEX, keep them ASCII, and use LuaTEX or XeTEX for Unicode. The preamble may register strings. Strings registered inside of `\include`d files are available as soon as the main file was run once. Words with digits are rules for `xlistings`.

Index of commands, keys and names

backend (key), 9
boundary (key), 2, 3, 5, 6, 9
codelink (key), 8
\CodeLinkAnchor, 2
\CodeLinkAnchorHook, 5
\CodeLinkBlock, 2
\CodeLinkFormat, 6
\CodeLinkGroup, 3
\CodeLinkLabel, 2
\CodeLinkLinkCommand, 9
\CodeLinkRef, 2
\CodeLinkRegister, 2
\CodeLinkSetup, 9
\CodeLinkStyle, 3
\CodeLinkTextFont, 2
command (key), 2, 3, 5
Counter (Python), 5
debug (key), 9
display (key), 3, 9
elements (Python), 5
group (key), 2, 3
highlight (key), 2, 3, 7
hyper (key), 9
hyperref (package), 1, 2, 9
l3keys2e (package), 1
label (key), 2, 3
language (key), 2, 3
links (key), 2, 3, 7
listings (package), 1, 8–10
minted (package), 1, 6, 9, 10
prefix (key), 2, 3
style (key), 2, 3
underline (key), 6, 9
underline-color (key), 7, 9
underline-offset (key), 7, 9
underline-text (key), 9
underline-text-color (key), 7, 9
underline-thickness (key), 7, 9
update (Python), 5
xcolor (package), 7
xlistings (package), 1, 6, 8, 10