Karnaj

Zeste de savoir

Présenter du code source dans un document LaTeX

3 avril 2021

Table des matières

Lorsque nous écrivons un document, il peut nous arriver de vouloir illustrer un propos par du code. Et comme d'habitude, nous pouvons trouver plusieurs *packages* pour nous aider à faire ce travail. L'un des plus connus est le *package* **listings**, mais dans ce tutoriel, nous apprendrons à le faire en utilisant le *package* **minted** . Les différences entre ces deux *packages* seront vues, ce qui permettra de choisir celui qui convient le mieux à chaque utilisation.

Prérequis
Connaissance des bases du LaTeX (un tutoriel est disponible ici).
Objectifs
Découvrir un moyen efficace de composer des codes sources en LaTeX.

1. Le texte préformaté et minted

1.1. Le texte préformaté

Pour écrire du code source, il nous faut écrire du code qui ne sera pas interprété par LaTeX et sera affiché tel qu'il est présenté (espaces, tabulations, sauts de lignes, etc.). De base, LaTeX propose la commande `verb` et l'environnement `verbatim` qui permettent de présenter du texte préformaté. Nativement, nous pouvons donc faire ceci.

```
\begin{document}
   Ici, on va montrer ce qu'il y a entre notre
       \verb|\begin{document}| et notre \verb|\end{document}|.
\begin{verbatim}
\begin{itemize}
   \item Élément.
   \item Un autre.
\end{itemize}
\end{verbatim}
   On voit bien dans ce document que \verb|\verb| et
       \verb|verbatim| font le travail voulu.
\end{document}
```

Le texte préformaté est traditionnellement présenté dans une police à chasse fixe (c'est-à-dire que tous les caractères ont la même largeur), `\verb` et `verbatim` ne font pas exception. Cependant, le code présenté avec `verbatim` n'est pas coloré, et nous n'avons pas la possibilité de numéroter les lignes et en fait de personnaliser l'affichage du code. C'est pourquoi plusieurs *packages* viennent compléter `verbatim`.

1.2. Pourquoi minted ?

Comme nous venons de le dire, plusieurs *packages* nous permettent de présenter du code. L'un des plus connu est le *package* **listings** et ce n'est pas pour rien. Il existe depuis longtemps, est mûr et éprouvé, fait très bien son travail, est très personnalisable, et c'est facile de trouver de la documentation à son sujet. En clair, il a tout ce que nous voulons… Cependant, dans ce tutoriel, nous allons apprendre à utiliser le *package* minted.

? Pourquoi apprendre à utiliser minted quand listings fait déjà ce qu'il faut et le fait bien? Quels sont les avantages de minted?

Le *package* listings a en effet beaucoup d'avantages, mais minted est plus simple à configurer, en particulier sur la coloration syntaxique (et produit des codes mieux formatés en général). Ainsi, alors qu'avec listings, il faut choisir la couleur de chaque type de mots (nous choisissons la couleur des chaînes de caractères, la couleur des mots-clés, la couleur des commentaires, etc.), avec minted, il y a juste à choisir un style et le code aura ce style.

i Voici deux documents , l'un ou le code a été formaté avec minted et l'autre avec listings. Nous voyons notamment qu'ici, minted a une gestion plus fine des éléments et de la coloration.

Pour ce faire, minted s'appuie sur un module Python, **Pygments** . Ce module gère plus de 300 langages différents et une bonne vingtaine de styles. Autant dire que nous avons le choix.

Bien sûr, cela a aussi quelques inconvénients: si nous voulons de la coloration pour un langage exotique (ou que nous venons de créer) que `Pygments` ne connaît pas, la configuration sera plus compliquée là où avec listings il suffirait d'indiquer la liste des mots-clés. Le fait que minted dépende de Python et de Pygments peut aussi être un frein à son utilisation pour certains.

1.3. Installer minted

Comme nous venons de le dire, minted s'appuie sur Python et son module `Pygments`. Il nous faut donc les installer. Pour ceux qui n'utilisent pas Python, cette page explique comment l'installer. Après l'avoir installé, installons `Pygments`. Pour cela, ouvrons un terminal et tapons la commande suivante.

```
pip install -U --user pygments
```

Une fois les opérations terminées, vérifions que pygments est bien installé en compilant un premier code qui utilise minted.

```
\documentclass[french]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}

\usepackage{babel}
\usepackage{minted}

\begin{document}
   Document de test de minted.
\end{document}
```

Puisque minted a besoin d'appeler Python, nous ne pouvons pas utiliser la commande `pdflatex` (ou `xelatex`, etc.) seule, il nous faut l'utiliser avec l'option `-shell-escape`. Ainsi, pour compiler notre document, nous utiliserons cette ligne de commande.

```
pdflatex -shell-escape fichier.tex
```

Bien sûr, `pdflatex` peut être changé par `xelatex` ou encore `lualatex`, etc.

Si nous oublions cette option, nous obtiendrons l'erreur « *You must invoke LaTeX with the -shell-escape flag* ». Si `pygments` n'est pas installé, nous obtiendrons l'erreur « *You muste have 'pygmentize' installed to use this package* ».

Une bonne solution pour régler ce problème est d'utiliser un outil comme LaTeXmk pour se faciliter la compilation de documents.

1.3.1. Windows

Pour ne pas oublier cette option, nous pouvons créer une nouvelle commande pour ne pas voir à taper cette option à chaque fois. Sous Windows, nous pouvons créer un fichier `.bat` avec ce contenu (nous pouvons par exemple l'appeler `mpdflatex.bat` pour «minted pdflatex».

```
pdflatex -shell-escape %*
```

Nous le plaçons alors dans le dossier où sont placés les exécutables `latex`, `pdflatex` et autres et pour compiler, nous pouvons maintenant écrire `mpdflatex fichier.tex`.

1.3.2. Linux et OSX

Sous Linux, il est aussi possible de créer une commande (ou d'utiliser les capacités de certains éditeurs de texte) pour ne pas avoir à tout taper à chaque fois.

1.3.3. TeXmaker

Sous Texmaker, il nous faut aller dans le menu «Options» → «Configurer Texmaker» et rajouter l'option à la commande qui nous intéresse (PdfLaTeX, XeLaTeX, etc.). Par exemple, pour PdfLaTeX, on a cette ligne.

```
pdflatex -synctex=1 -interaction=nonstopmode %.tex
```

Elle devient la suivante.

```
pdflatex -synctex=1 -interaction=nonstopmode --shell-escape %.tex
```

1.3.4. Emacs et AucTeX

Pour utiliser `minted` facilement avec Emacs et AucTex, on peut se rapporter à cette réponse sur Stackoverflow . Elle permet de modifier la façon dont LaTeX est appelé par AucTex pour lui rajouter l'option `-shell-escape`.

1.3.5. Vim et vim-latex-live-preview

La plupart des gens qui utilisent Vim pour faire du LaTeX utilisent également vim-latex-live-preview. Pour le faire fonctionner avec `minted`, il faudra changer la commande de compilation. Pour cela, nous allons modifier la variable spécifiant le moteur utilisé pour la compilation en lui indiquant de compiler avec des options.

```
let g:livepreview_engine = pdflatex . -interaction=nonstopmode
    --shell-escape
```

Nous sommes maintenant prêts à apprendre à utiliser minted.

2. Écrire du code

2.1. Utiliser minted

2.1.1. Bloc de code

Comme nous l'avons dit, utiliser minted est très simple. Pour écrire un bloc de code, il nous suffit d'utiliser l'environnement `minted`, qui prend en paramètre le langage du bloc de code (nous pouvons voir la liste des langages sur le site de Pygments ou en utilisant la commande `pygmentize -L lexers`). Voici donc un premier exemple de code.

```latex
\documentclass[french]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}

\usepackage{babel}
\usepackage{minted}

\begin{document}
   Voici un bloc de code en LaTeX.

\begin{minted}{latex}
En LaTeX, on écrit des maths en ligne entre deux signes dollars.
Par exemple, $1 + 3^2 = 10$.
Pour les maths hors-texte, voici un exemple.
\[
   1 + 3^2.
\]
\end{minted}
\end{document}
```

Compilons-le et ouvrons le document PDF obtenu. Nous remarquons déjà que la coloration syntaxique est bien au rendez-vous, et ce sans rien avoir eu à indiquer. Nous pouvons essayer d'autres langages, d'autres codes et voir comment ils sont composés.

Si nous voulons faire un bloc de code d'une ligne, nous pouvons utiliser la commande `mint` qui est alors un raccourci de l'environnement `minted`. Elle prend en paramètre le langage à utiliser

et s'utilise comme `\verb` c'est-à-dire que pour délimiter le code, nous utilisons deux fois un même caractère qui n'est pas présent dans le code.

```
Observons que nous avons a le même résultat.

\mint{latex}|En LaTeX \begin et \end servent pour les
    environnements.|

\begin{minted}{latex}
En LaTeX \begin et \end servent pour les environnements.
\end{minted}
```

2.1.2. Depuis un fichier

Plutôt que d'écrire le code dans le fichier LaTeX, il est possible d'afficher le code d'un fichier en utilisant la commande `inputminted` qui prend en paramètre le langage et le chemin du fichier. Par exemple, affichons le contenu du fichier `fichier.tex` que nous sommes en train d'écrire.

```
\inputminted{latex}{test.tex}
```

2.1.3. Code en mode en ligne

Et finalement, pour mettre du code en mode en ligne, nous allons utiliser la commande `\mintinline`. Elle prend en paramètre le langage à utiliser et le code qui peut être entre accolades, ou, comme avec `\mint` ou `\verb`, être autour d'une paire de caractères.

```
La commande \mintinline{latex}{\begin} ouvre un environnement.
```

Et un autre exemple où nous mettons du code dans un titre…

```
\section{\mintinline{latex}{\section}}
```

… Qui ne fonctionne pas.

En fait, la commande `\mintinline` est une commande fragile[1]. Cela signifie que son utilisation peut poser problème lorsqu'elle est en argument d'une commande à arguments dits «mouvants» (nous n'allons pas nous attarder ici sur ce que cela signifie). Par exemple, l'argument de `section` (et des commandes de sectionnement en général) est mouvant.

Dans le cas où `mintinline` est un argument mouvant d'une autre commande , il nous faudra alors faire attention à la protéger en utilisant la commande `\protect`. Parfois, cela ne fonctionnera pas; une autre solution, mais qui n'utilise plus minted et n'a pas la coloration syntaxique, est d'utiliser la commande `\texttt` pour obtenir une police à chasse fixe. Le problème est alors que nous ne pouvons pas utiliser les caractères spéciaux directement. Ce n'est pas une solution officielle, mais c'est certainement la meilleure à ce jour (avec la version `2.4` de minted).

Voici donc un code qui fonctionne.

```
%
\section{\texttt{\textbackslash section}}
```

Une autre solution (un peu plus compliquée et qui ne sera pas exposée ici) proposée sur le Github du projet est d'utiliser la commande `\newsavebox`.

2.2. Les différents styles

Maintenant que nous connaissons les bases de minted, nous allons voir comment changer de style. Pour cela nous utilisons la commande `\usemintedstyle`. Elle prend en paramètre un style et l'applique à tous les codes (jusqu'à ce qu'on change à nouveau de style). La liste des styles est disponible avec la commande `pygmentize -L styles`. Nous pouvons voir à quoi ils ressemblent en faisant des tests sur le site de Pygments .

Testons par exemple les styles `monokay` et `xcode` (dans le code qui suit, `text.tex` est le nom du fichier courant).

```
\usemintedstyle{monokay}

\begin{document}
   Avec le style \mintinline{latex}{monokay}.
   \inputminted{latex}{test.tex}
   \usemintedstyle{xcode}
   Avec le style \mintinline{latex}{xcode}.
   \inputminted{latex}{test.tex}
\end{document}
```

La commande `\usemintedstyle` prend aussi en paramètre facultatif le langage auquel doit s'appliquer le style choisi. Cela permet alors de préciser un style différent pour plusieurs langages. Nous pouvons alors choisir un style pour du Python et un autre pour du Ruby.

```
\usemintedstyle[python]{vim}
\usemintedstyle[ruby]{emacs}

\mint{python}|print('Hello word')|
```

```
5 \mint{ruby}|puts 'Hello word'|
```

Nous voulons choisir un même style pour plusieurs langages, en séparant les langages par une virgule dans le paramètre facultatif.

```
1 \usemintedstyle[python, ruby]{emacs}
```

Notons également qu'il est possible de choisir un style pour un bout de code en particulier. En effet, les commandes `mint` et `mintinline` ainsi que l'environnement `minted` prennent en paramètre facultatif des options. L'option `style` permet de choisir le style.

```
1 \mintinline[style = xcode]{python}|print("Hello word")|
```

2.3. Flottants

Tout comme les tableaux, nous pouvons rendre un bloc de code flottant en utilisant l'environnement `listing` qui fonctionne comme l'environnement `figure`. Bien sûr, nous pouvons y mettre une légende et y faire référence, et nous disposons même de la commande `\listoflistings` pour obtenir la liste des codes sources.

```
1  Voici le code du document.
2
3  \begin{listing}
4     \inputminted{latex}{test.tex}
5     \caption{Code du document.}
6     \label{lst:code_1}
7  \end{listing}
8
9  Le code du document (voir code \ref{lst:code_1} page
       \pageref{lst:code_1}
10 a pu être écrit facilement grâce à minted.
11
12 \listoflistings
```

C'est une bonne pratique de construire le nom des références en le faisant précéder par `lst` pour rappeler qu'il s'agit d'un code.

Si nous voulons la possibilité d'avoir une légende sans utiliser de flottants, nous pouvons utiliser le *package* **caption** . Avec lui, le code précédent se transforme de cette manière.

```latex
\usepackage{caption}
\usepackage{minted}

\begin{document}
Voici le code du document.

\begin{center}
   \inputminted{latex}{test.tex}
   \captionof{listing}{Code du document.}
   \label{lst:code_1}
\end{center}

Le code du document (voir code \ref{lst:code_1} page
    \pageref{lst:code_1}
a pu être écrit facilement grâce à minted.

\listoflistings
\end{document}
```

Nous remarquons que la liste des blocs de codes est toujours correcte. Le *package* caption peut tout aussi bien être chargé après minted, cela ne pose aucun souci.

2.3.1. Changer les noms

Nous pouvons changer le nom affiché devant les légendes (par défaut c'est «*Listing*») en redéfinissant la commande `\listingscaption`. Nous pouvons alors afficher «Code source» à la place en plaçant ceci dans notre préambule.

```latex
\renewcommand{\listingscaption}{Code source}
```

De même, nous pouvons changer le titre de la liste des codes sources en redéfinissant la commande `\listoflistingscaption`.

```latex
\renewcommand{\listoflistingscaption}{Liste des codes sources}
```

3. Personnalisation

En plus du style, nous pouvons personnaliser les blocs de code que nous affichons de plusieurs manières (numéro de ligne, cadre, etc.).

1. Plus d'informations ici (en anglais)

3.1. Les options

Nous pouvons choisir plusieurs options grâce à la commande `\setminted` qui prend en paramètre un langage et les options que nous voilons pour les codes de ce langage. Les options choisies avec cette option s'appliquent aux blocs de code, mais aussi au code en ligne. Cependant, si nous voulons spécifier une option pour le code en ligne, nous pouvons utiliser la commande `\setmintedinline` qui fonctionne de la même manière.

Les options sont sous la forme `clé=valeur`, la valeur pouvant être omise s'il s'agit de `true`. Par exemple, l'option `mathscope` permet, si elle est mise à `true` d'activer le mode mathématique dans les commentaires des codes présentés.

```latex
\setminted{mathescape}

\begin{minted}{latex}
\[
    \exp(0) + 1 = 2 % Affiche $\exp(0) + 1 = 2$
\]
\end{minted}
```

Nous pouvons, comme avec `\usemintedstyle`, indiquer en paramètre facultatif le ou les langages pour lesquels nous voulons ces options. En fait, `style` est aussi une option et utiliser `\usemintedstyle` équivaut à utiliser `\setminted` pour choisir la valeur de `style`.

De plus, l'environnement `minted`, tout comme les commandes `\mint` et `\mintinline` prennent aussi des options en paramètre facultatif. Cela permet alors de choisir des options pour un bloc de code particulier.

```latex
\begin{minted}[mathescape]{latex}
\[
    \exp(0) + 1 = 2 % Affiche $\exp(0) + 1 = 2$
\]
\end{minted}
```

Maintenant, voyons quelques types d'options.

3.1.1. Les césures

Nous avons une bonne vingtaine d'options concernant les césures, c'est-à-dire la manière dont une ligne d'un code est coupée si nous sommes à la fin d'une ligne de notre document. La plus important d'entre elle, `breaklines` est un booléen, qui vaut `false` par défaut et permet s'il vaut `true` d'autoriser LaTeX à couper une ligne trop longue.

Normalement, les coupures ne se font qu'aux espaces, mais ceci peut être changé grâce à l'option `breakafter` qui prend pour valeur les caractères après lesquels la coupure est autorisée. Notons que les caractères spéciaux doivent être échappés (pour indiquer une accolade ouvrante, ce sera `\{`). Nous pouvons autoriser les coupures n'importe où grâce à l'option `breakanywhere`, un booléen qui vaut `false` par défaut. Bien sûr, ces deux options ne font effet que si `breaklines` est activée.

```
\begin{minted}[breaklines, breakafter=-\{\}\%+-*]{latex}
  En LaTeX, on écrit des maths facilement. $23 - 10 * 4 = 23 - 40 =
     -17$.
\end{minted}
```

Nous remarquons ici qu'un symbole est inséré au début de la seconde ligne (après la coupure). Pour changer ce symbole, nous utiliserons l'option `breaksymbol` (ou l'option `breaksymbol left` qui est strictement équivalente), sachant que pour n'avoir aucun symbole, il nous suffit d'utiliser `breaksymbol={}` ou `breaksymbol=`.

```
\begin{minted}[breaklines, breakafter=-\{\}\%+-*, breaksymbol =
    ]{latex}
  En LaTeX, on écrit des maths facilement. $23 - 10 * 4 = 23 - 40 =
     -17$.
\end{minted}
```

Quand une ligne est coupée, il peut-être intéressant de garder le reste de la ligne au même niveau d'indentation que le début. Pour ce faire, nous allons utiliser l'option `breakautoindent`, un booléen valant `true` par défaut et permettant d'indenter le reste de la ligne. Dans la même veine, l'option `breakindent` (qui est une longueur) permet de spécifier la longueur de l'indentation à rajouter au début de chaque ligne coupée (elle permet donc de gérer cela plus précisément qu'avec `breakautoindent`).

```
\begin{minted}[breaklines, breakafter=-+*/., breaksymbol = ,
    breakindent = 3em]{python}
def f(x):
    print("La valeur de x + 4 est {}, celle de x + 5 est
        {}".format(x + 4, x + 5))
end
\end{minted}
```

3.1.2. L'indentation

Pour continuer avec l'indentation, voici une option utile pour placer l'indentation du code LaTeX dans le code que nous souhaitons afficher **sans** afficher cette indentation.

```latex
\begin{minted}{latex}
   \[
      2 * 3 = 2 + 2 + 2 = 4 + 2 = 6.
   \]
\end{minted}
```

Ici les six espaces de l'indentation dues à notre code LaTeX seront aussi affichés. Pour ne pas les afficher, nous allons utiliser l'option `gobble` qui est un entier `n`. Ainsi les `n` premiers caractères de chaque ligne de code seront supprimés (ici nous voulons `n = 6`). En fait, nous pouvons même faire mieux grâce à l'option `autogobble`, un booléen, qui permet de supprimer tous les espaces blancs (espaces et tabulations) communs au début de chaque ligne. Notre code précédent se transforme de cette manière.

```latex
\begin{minted}[autogobble]{latex}
   \[
      2 * 3 = 2 + 2 + 2 = 4 + 2 = 6.
   \]
\end{minted}
```

3.1.3. Les cadres

Nous pouvons bien entendu encadrer nos codes (bien sûr, ceci ne s'applique pas aux codes en ligne). Ceci se fait grâce à l'option `frame` qui peut prendre beaucoup de valeurs.

- `none`, qui est la valeur par défaut, ne crée pas de cadre.
- `single` crée un cadre complet.
- `leftline` crée uniquement la ligne gauche du cadre.
- `topline` crée uniquement la ligne en haut du cadre.
- `bottomline` crée uniquement la ligne en bas du cadre.
- `topline` crée uniquement la ligne en haut du cadre.
- `lines` crée uniquement les lignes horizontales du cadre.

```latex
\begin{minted}[autogobble, frame = single]{latex}
   \[
      2 * 3 = 2 + 2 + 2 = 4 + 2 = 6.
   \]
\end{minted}
\begin{minted}[autogobble, frame = lines]{latex}
   \[
      2 * 3 = 2 + 2 + 2 = 4 + 2 = 6.
   \]
\end{minted}
```

3.1.4. Numérotation

Pour numéroter les lignes de code, nous utiliserons l'option `linenos` (booléen valant `true` si nous voulons activer la numérotation). Chaque ligne de code est alors numéroté, la numérotation commençant à zéro pour chaque bloc de code (bien entendu, la numérotation n'a pas lieu pour les bouts de code dans le texte. Avec l'option `\numbers` (qui vaut `auto` par défaut), nous pouvons choisir de quel côté la numérotation est affichée (`left`, `right` ou `both` pour l'afficher des deux côtés).

De plus, nous pouvons choisir le premier nombre qui servira à numéroter les lignes grâce au paramètre `firstnumber` qui vaut `auto` par défaut. Nous pouvons lui donner comme valeur le nombre par lequel nous voulons commencer, mais aussi `last`, auquel cas, la numérotation continuera celle du précédent bloc de code.

```
\begin{minted}[autogobble, linenos, frame = single]{latex}
   \[
      2 * 3 = 2 + 2 + 2 = 4 + 2 = 6.
   \]
\end{minted}

\begin{minted}[autogobble, linenos, firstnumber = last]{latex}
   \[
      2 * 3 = 2 + 2 + 2 = 4 + 2 = 6.
   \]
\end{minted}
```

Ici, la numérotation du second bloc de code continue celle du premier. Nous remarquons de plus que la numérotation se fait en dehors du cadre.

Il y a plusieurs autres options (voir la documentation pour en savoir plus).

3.2. Un environnement pour un langage

Nous avons la possibilité de définir une commande pour un langage en utilisant la commande `\newminted`. Elle prend en paramètre un langage et les options que nous voulons.

```
\newminted{latex}{frame = single, style = xcode, autogobble}
```

Ici, nous indiquons que nous voulons créer un raccourci pour les code LaTeX et qu'avec ce raccourci, le code sera encadré et aura le style `xcode`. La commande `\newminted` crée alors l'environnement `latexcode` (et donc `<langage>code` pour un autre langage). Nous pouvons alors l'utiliser et observer le résultat.

```latex
\begin{latexcode}
   \begin{document}
      Un document LaTeX
   \end{document}
\end{latexcode}
```

La commande `\newminted` permet alors de ne pas spécifier les options à chaque usage de `minted` (tout comme `\setminted`) et de ne pas avoir à spécifier le langage à chaque fois.

La commande `\newminted` crée aussi un environnement étoilé `code<langage>` qui prend en second paramètre les options supplémentaires que nous pourrions vouloir pour notre code. Par exemple, si dans un document nous voulons tout nos codes LaTeX encadrés et avec le style `xcode`, et que pour un code particulier, nous voulons numéroter les lignes, l'environnement étoilé est utile.

```latex
\begin{latexcode*}{linenos}
   \begin{document}
      Un document LaTeX
   \end{document}
\end{latexcode*}
```

i Il est possible de choisir le nom de l'environnement créé en le spécifiant en paramètre facultatif de `\newminted`.

```latex
\newminted[latex]{latex}{frame = single, style = xcode}
```

Nous pouvons aussi créer des commandes-raccourcis pour `\mint`, `\mintinline` et `\inputminted`, en utilisant respectivement les commandes `\newmint`, `\newmintinline` et `\newmintedfile` qui fonctionnent de la même manière que `\newminted`. Les noms par défaut des commandes raccourcis sont alors:

— `\<langage>` pour `\mint`.
— `\<langage>inline` pour `\mintinline`.
— `<langage>file` pour `\newmintedfile`.

Tout cela nous permet de construire un document LaTeX très flexible.

3.3. Les options de minted

3.3.1. Numérotation

Le *package* minted a aussi des options (donc à préciser lorsqu'il est chargé). Par exemple, par défaut le compteur pour la numérotation des flottants (donc de l'environnement `listing`)

n'est jamais remis à zéro. Nous pouvons utiliser l'option `chapter` ou l'option `section` pour le remettre à zéro à chaque chapitre ou à chaque section et avoir une numérotation adaptée. Essayons par exemple ce code sans rien, puis avec l'option `chapter`, et enfin avec l'option `section`.

```latex
\newmint[latex]{latex}{frame = single, style = xcode}

\begin{document}
   \chapter{Un}
      \begin{listing}
         \mint{latex}|\begin{document}|
         \caption{Légende}
      \end{listing}

   \chapter{Deux}
      \section{sec}
         \begin{listing}
            \mint{latex}|\begin{document}|
            \caption{Légende}
         \end{listing}

         \begin{listing}
            \mint{latex}|\begin{document}|
            \caption{Légende}
         \end{listing}
      \section{sec}
         \begin{listing}
            \mint{latex}|\begin{document}|
            \caption{Légende}
         \end{listing}
\end{document}
```

3.3.2. Document final

Nous pouvons également choisir entre l'option `draft` et l'option `final` (donc entre un document brouillon et un document final), l'option par défaut étant `final`. L'option `draft` permet d'obtenir une compilation plus rapide. Pour cela, Pygmentize n'est pas utilisé (donc il n'y a pas de coloration syntaxique). Cela permet alors d'utiliser `pdflatex` sans `-shell-escape`. De même, en mode brouillon, `autogobble` ne fonctionne pas.

L'option `draft` peut être utile pour faire des tests, mais aussi dans le cas où nous obtenons des erreurs que nous voulons traquer et que l'apparence du code ne nous intéresse pas. Pour faire un ECM, cela peut également être utile.

3.3.3. Options de cache

Lorsque minted croise un bout de code à transformer, il appelle Pygments pour qu'il lui donne le code avec la coloration syntaxique appropriée. Si ces appels se faisaient à chaque compilation (et ce même pour les codes qui n'ont pas été modifiés) le temps de compilation augmenterait ce qui serait problématique dans le cas de très gros documents. Heureusement, l'option `cache` (un booléen valant `true` par défaut) permet de mettre ces données en cache (dans un dossier qu'il crée dans le dossier du projet). Puisque ce booléen vaut `true` par défaut, nous n'avons rien à faire pour activer le cache.

Les informations placées en cache sont par défaut dans le dossier `_minted-<jobname>` (par exemple, si je compile `file.tex`, ce sera le dossier `_minted-file`. Ce dossier peut être modifié avec l'option `cachedir`.

3.3.3.1. Compilation sans `-shell-escape` Parfois, nous ne maîtrisons pas le processus de compilation et ne pouvons donc pas rajouter l'option `-shell-escape` à la commande de compilation. C'est typiquement le cas lors de la soumission d'articles, les éditeurs voulant généralement recompiler eux-mêmes le document. Pour remédier à ce problème, nous allons utiliser un couple d'options, `finalizecache` et `frozencache` qui valent `false` par défaut.

L'option `finalizecache` permet d'indiquer que les codes de notre document sont définitifs. Ils seront alors totalement mis en cache et pourront être utilisés par la suite. C'est une préparation à la compilation sans `-shell-escape`. L'option `frozencache` est à utiliser après qu'un code a été mis en cache avec `finalizecache`. Lorsque cette option est activée, nous n'avons plus besoin d'utiliser `-shell-escape`, les données en cache seront utilisées pour mettre en page le code. Nous allons donc agir en deux étapes.

1. Compiler le document avec `-shell-escape` en ayant pris soin d'activer l'option `finalizecache`.
2. Compiler le document **sans** `-shell-escape` en remplaçant l'option `finalizecache` par l'option `frozencache` (bien sûr, le dossier des fichiers mis en cache doit alors être présent).

Nous pouvons maintenant mettre du beau code dans nos documents. Bien sûr, nous n'avons pas vu toutes les options disponibles. Pour les voir toutes, il faut se référer à la documentation . Notons de plus que nous avons la possibilité de créer nos propres styles en nous basant sur Pygments.

Merci à @TD, à @Heziode pour leurs retours durant la bêta et à @Saroupille pour avoir validé ce tutoriel.

Liste des abréviations

ECM Exemple complet minimal. 15