L'extension LaTeX piton *

F. Pantigny
fpantigny@wanadoo.fr

24 mars 2024

Résumé

L'extension piton propose des outils pour composer des codes informatiques en Python, OCaml, C et SQL avec une coloration syntaxique en utilisant la bibliothèque Lua LPEG. L'extension piton nécessite l'emploi de LuaLaTeX.

1 Présentation

L'extension piton utilise la librairie Lua nommée LPEG [1] pour « parser » le code Python, OCaml, C ou SQL et le composer avec un coloriage syntaxique. Comme elle utilise le Lua de LuaLaTeX, elle fonctionne uniquement avec `lualatex` (et ne va pas fonctionner avec les autres moteurs de compilation LaTeX, que ce soit `latex`, `pdflatex` ou `xelatex`). Elle n'utilise aucun programme extérieur et la compilation ne requiert donc pas `--shell-escape`. La compilation est très rapide puisque tout le travail du parseur est fait par la librairie LPEG, écrite en C.

Voici un exemple de code Python composé avec l'environnement `{Piton}` proposé par piton.

```
from math import pi

def arctan(x,n=10:int):
    """Calcule la valeur mathématique de arctan(x)

    n est le nombre de termes de la somme
    """
    if x < 0:
        return -arctan(-x) # appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x)
        (on a utilisé le fait que arctan(x) + arctan(1/x) = π/2 pour x > 0) [2]
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

*Ce document correspond à la version 2.7 de piton, à la date du 2024/03/24.

1. LPEG est une librairie de capture de motifs (*pattern-matching* en anglais) pour Lua, écrite en C, fondée sur les PEG (*parsing expression grammars*) : `http://www.inf.puc-rio.br/~roberto/lpeg/`

2. Cet échappement vers LaTeX a été obtenu en débutant par `#>`.

2 Installation

L'extension piton est composée de deux fichiers : `piton.sty` et `piton.lua` (le fichier LaTeX `piton.sty` chargé par `\usepackage` va à son tour charger le fichier `piton.lua`). Les deux fichiers doivent être présents dans un répertoire où LaTeX pourra les trouver, de préférence dans une arborescence `texmf`. Le mieux reste néanmoins d'installer piton avec une distribution TeX comme MiKTeX, TeX Live ou MacTeX.

3 Utilisation de l'extension

L'extension piton n'est utilisable qu'avec LuaLaTeX : si un autre moteur de compilation (comme `latex`, `pdflatex` ou `xelatex`) est utilisé, une erreur fatale sera levée.

3.1 Choix du langage

L'extension piton prend en charge quatre langages informatiques : Python, OCaml, SQL et C, ou plutôt C++. Il prend aussi en charge un langage minimal appelé « `minimal` » : cf. p. 30.

Par défaut, le langage est Python.

On peut changer de langage avec la clé `language` de `\PitonOptions` :

```
\PitonOptions{language = C}
```

Pour les développeurs, précisons que le nom du langage courant est stocké (en minuscules) dans la variable publique L3 nommée `\l_piton_language_str`.

Dans la suite de ce document, on parlera de Python mais les fonctionnalités s'appliquent aussi aux autres langages.

3.2 Chargement de l'extension

L'extension piton se charge simplement avec `\usepackage{piton}`.

Si, à la fin du prambule, l'extension xcolor n'a pas été chargée (par l'utilisateur ou par une extension chargée dans le préambule), piton charge l'extension xcolor avec `\usepackage{xcolor}`, c'est-à-dire sans aucune option. L'extension piton ne charge pas d'autre extension.

3.3 Les commandes et environnements à la disposition de l'utilisateur

L'extension piton fournit plusieurs outils pour composer du code Python : les commandes `\piton`, l'environnement `{Piton}` et la commande `\PitonInputFile`.

— La commande `\piton` doit être utilisée pour composer de petits éléments de code à l'intérieur d'un paragraphe. Par exemple :

```
\piton{def carré(x): return x*x}
```

```
def carré(x): return x*x
```

La syntaxe et les particularités de la commande sont détaillées ci-après.

— L'environnement `{Piton}` doit être utilisé pour composer des codes de plusieurs lignes. Comme cet environnement prend son argument selon un mode verbatim, il ne peut pas être utilisé dans l'argument d'une commande LaTeX. Pour les besoins de personnalisation, il est possible de définir de nouveaux environnements similaires à `{Piton}` en utilisant la commande `\NewPitonEnvironment` : cf. partie 4.3 p. 8.

— La commande `\PitonInputFile` doit être utilisée pour insérer et composer un fichier externe.

Il est possible de n'insérer qu'une partie de ce fichier : cf. partie 5.2, p. 10.

La clé `path` de la commande `\PitonOptions` permet de spécifier un chemin pour le fichier à inclure.

3.4 La syntaxe de la commande \piton

La commande `\piton` possède en fait une syntaxe double. Elle est peut être utilisée comme une commande standard de LaTeX prenant son argument entre accolades (`\piton{...}`), ou bien selon la syntaxe de la commande `\verb` où l'argument est délimité entre deux caractères identiques (par ex. : `\piton|...|`). On détaille maintenant ces deux syntaxes.

— Syntaxe `\piton{...}`

Quand son argument est donné entre accolades, la commande `\piton` ne prend pas son argument en mode verbatim. Les points suivants doivent être remarqués :

— plusieurs espaces successives sont remplacées par une unique espace, ainsi que les retours à la ligne
mais la commande `\␣` est fournie pour forcer l'insertion d'une espace ;

— il n'est pas possible d'utiliser le caractère `%` à l'intérieur,
mais la commande `\%` est fournie pour insérer un `%` ;

— les accolades doivent apparaître par paires correctement imbriquées,
mais les commandes `\{` et `\}` sont aussi fournies pour insérer des accolades individuelles ;

— les commandes LaTeX [3] sont complètement développées sans être exécutées
et on peut donc utiliser `\\` pour insérer une contre-oblique.

Les autres caractères (y compris `#`, `^`, `_`, `&`, `$` et `@`) doivent être insérés sans contre-oblique.

Exemples :

```
\piton{ma_chaîne = '\\n'}                    ma_chaîne = '\n'
\piton{def pair(n): return n\%2==0}          def pair(n): return n%2==0
\piton{c="#"    # une affectation }          c="#" # une affectation
\piton{c="#" \ \ \ # une affectation }       c="#"    # une affectation
\piton{my_dict = {'a': 3, 'b': 4}}           my_dict = {'a': 3, 'b': 4}
```

La commande `\piton` avec son argument entre accolades peut être utilisée dans les arguments des autres commandes LaTeX. [4]

— Syntaxe `\piton|...|`

Quand la commande `\piton` prend son argument entre deux caractères identiques, cet argument est pris *en mode verbatim*. De ce fait, avec cette syntaxe, la commande `\piton` ne peut *pas* être utilisée dans l'argument d'une autre fonction.

Exemples :

```
\piton|ma_chaîne = '\n'|                     ma_chaîne = '\n'
\piton!def pair(n): return n%2==0!           def pair(n): return n%2==0
\piton+c="#"    # une affectation +          c="#"     # une affectation
\piton?my_dict = {'a': 3, 'b': 4}?           my_dict = {'a': 3, 'b': 4}
```

4 Personnalisation

Concernant la fonte de caractères utilisée dans les listings produits par l'extension piton, il s'agit simplement de la fonte mono-chasse courante (piton utilise simplement en interne la commande LaTeX standard `\ttfamily`). Pour la changer, le mieux est d'utiliser `\setmonofont` de fontspec.

3. Cela s'applique aux commandes commençant par une contre-oblique `\` mais également aux caractères actifs, c'est-à-dire ceux de catcode 13.

4. La commande `\piton` peut par exemple être utilisée dans une note de bas de page. Exemple : `s = 'Une chaîne'`.

4.1 Les clés de la commande \PitonOptions

La commande `\PitonOptions` prend en argument une liste de couples *clé=valeur*. La portée des réglages effectués par cette commande est le groupe TeX courant. [5]
Ces clés peuvent aussi être appliquées à un environnement `{Piton}` individuel (entre crochets).

— La clé `language` spécifie le langage informatique considéré (la casse n'est pas prise en compte). Cinq valeurs sont possibles : `Python`, `OCaml`, `C`, `SQL` et `minimal`. La valeur initiale est `Python`.

— La clé `path` indique un chemin où seront cherchés les fichiers inclus par `\PitonInputFile`.

— La clé `gobble` prend comme valeur un entier positif n : les n premiers caractères de chaque ligne sont alors retirés (avant formatage du code) dans les environnements `{Piton}`. Ces n caractères ne sont pas nécessairement des espaces.

— Quand la clé `auto-gobble` est activée, l'extension piton détermine la valeur minimale n du nombre d'espaces successifs débutant chaque ligne (non vide) de l'environnement `{Piton}` et applique `gobble` avec cette valeur de n.

— Quand la clé `env-gobble` est activée, piton analyse la dernière ligne de l'environnement, c'est-à-dire celle qui contient le `\end{Piton}` et détermine si cette ligne ne comporte que des espaces suivis par `\end{Piton}`. Si c'est le cas, piton calcule le nombre n de ces espaces et applique `gobble` avec cette valeur de n. Le nom de cette clé vient de *environment gobble* : le nombre d'espaces à retirer ne dépend que de la position des délimiteurs `\begin{Piton}` et `\end{Piton}` de l'environnement.

— La clé `write` prend en argument un nom de fichier (avec l'extension) et écrit le contenu [6] de l'environnement courant dans ce fichier. À la première utilisation du fichier par piton, celui-ci est effacé.

— **Nouveau 2.5** La clé `path-write` indique un chemin où seront écrits les fichiers écrits par l'emploi de la clé `write` précédente.

— La clé `line-numbers` active la numérotation des lignes (en débordement à gauche) dans les environnements `{Piton}` et dans les listings produits par la commande `\PitonInputFile`.

Cette clé propose en fait plusieurs sous-clés.

— La clé `line-numbers/skip-empty-lines` demande que les lignes vides (qui ne contiennent que des espaces) soient considérées comme non existantes en ce qui concerne la numérotation des lignes (si la clé `/absolute` est active, la clé `/skip-empty-lines` n'a pas d'effet dans `\PitonInputFile`). La valeur initiale de cette clé est `true` (et non `false`). [7]

— La clé `line/numbers/label-empty-lines` demande que les labels (c'est-à-dire les numéros) des lignes vides soient affichés. Si la clé `/skip-empty-lines` est active, la clé `/label-empty-lines` est sans effet. La valeur initiale de cette clé est `true`.

— La clé `line-numbers/absolute` demande, pour les listings générés par `\PitonInputFile`, que les numéros de lignes affichés soient absolus (c'est-à-dire ceux du fichier d'origine). Elle n'a d'intérêt que si on n'insère qu'une partie du fichier (cf. part 5.2, p. 10). La clé `/absolute` est sans effet dans les environnements `{Piton}`.

— La clé `line-numbers/resume` reprend la numérotation là où elle avait été laissée au dernier listing. En fait, la clé `line-numbers/resume` a un alias, qui est `resume` tout court.

— La clé `line-numbers/start` impose que la numérotation commence à ce numéro.

— La clé `line-numbers/sep` est la distance horizontale entre les numéros de lignes (insérés par `line-numbers`) et les lignes du code informatique. La valeur initiale est 0.7 em.

Pour la commmodité, un dispositif de factorisation du préfixe `line-numbers` est disponible, c'est-à-dire que l'on peut écrire :

```
\PitonOptions
  {
    line-numbers =
```

5. On rappelle que tout environnement LaTeX est, en particulier, un groupe.
6. En fait, il ne s'agit pas exactement du contenu de l'environnement mais de la valeur renvoyée par l'instruction Lua `piton.get_last_code()` qui en est une version dans les surcharges de formatage LaTeX (voir la partie 6, p. 18).
7. Avec le langage Python, les lignes vides des *docstrings* sont prises en compte.

```
        {
          skip-empty-lines = false ,
          label-empty-lines = false ,
          sep = 1 em
        }
    }
```

— La clé `left-margin` fixe une marge sur la gauche. Cette clé peut être utile, en particulier, en conjonction avec la clé `line-numbers` si on ne souhaite pas que les numéros de ligne soient dans une position en débordement sur la gauche.

Il est possible de donner à la clé `left-margin` la valeur spéciale `auto`. Avec cette valeur, une marge est insérée automatiquement pour les numéros de ligne quand la clé `line-numbers` est utilisée. Voir un exemple à la partie 7.1 p. 19.

— La clé `background-color` fixe la couleur de fond des environnements `{Piton}` et des listings produits par `\PitonInputFile` (ce fond a une largeur que l'on peut fixer avec la clé `width` décrite ci-dessous). La clé `background-color` accepte une couleur définie « à la volée », c'est-à-dire que l'on peut écrire par exemple `background-color = [cmyk]{0.1,0.05,0,0}`

La clé `background-color` accepte aussi en argument une *liste* de couleurs. Les lignes sont alors coloriées de manière cyclique avec ces couleurs.

Exemple : `\PitonOptions{background-color = {gray!5,white}}`

— Avec la clé `prompt-background-color`, piton ajoute un fond coloré aux lignes débutant par le prompt « `>>>` » (et sa continuation « `...` ») caractéristique des consoles Python avec boucle REPL (*read-eval-print loop*). Pour un exemple d'utilisation de cette clé, voir la partie 8.2 p. 23.

— La clé `width` fixe la largeur du listing produit. Cette largeur s'applique aux fonds colorés spécifiés par les clés `background-color` et `prompt-background-color` et également quand une coupure automatique des lignes est demandée par `break-lines` (cf. 5.1.2, p. 9).

Cette clé peut prendre comme valeur une longueur explicite mais aussi la valeur spéciale `min`. Avec cette valeur, la largeur sera calculée à partir de la largeur maximale des lignes de code. Attention : l'usage de cette valeur spéciale `min` requiert deux compilations LuaLaTeX [8].

Pour un exemple d'utilisation de `width=min`, voir la partie 7.2 sur les exemples, p. 19.

— En activant la clé `show-spaces-in-strings`, les espaces dans les chaînes de caractères [9] sont matérialisés par le caractère ␣ (U+2423 : OPEN BOX). Bien sûr, le caractère U+2423 doit être présent dans la fonte mono-chasse utilisée. [10]

Exemple : `my_string = 'Très␣bonne␣réponse'`

— Avec la clé `show-spaces`, tous les espaces sont matérialisés (et aucune coupure de ligne ne peut plus intervenir sur ces espaces matérialisés, même si la clé `break-lines` [11] est active). Il faut néanmoins remarquer que les espaces en fin de ligne sont tous supprimés par piton. Les tabulations de début de ligne sont représentées par des flèches.

```
\begin{Piton}[language=C,line-numbers,auto-gobble,background-color = gray!15]
    void bubbleSort(int arr[], int n) {
        int temp;
        int swapped;
        for (int i = 0; i < n-1; i++) {
            swapped = 0;
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    temp = arr[j];
```

8. La largeur maximale est calculée lors de la première compilation, écrite sur le fichier `aux`, puis réutilisée lors de la compilation suivante. Certains outils comme `latexmk` (utilisé par Overleaf) effectuent automatiquement un nombre suffisant de compilations.

9. Pour le language Python, cela ne s'applique que pour les chaînes courtes, c'est-à-dire celles délimitées par `'` ou `"`. En OCaml, cela ne s'applique pas pour les *quoted strings*.

10. L'extension piton utilise simplement la fonte mono-chasse courante. Pour la changer, le mieux est d'utiliser `\setmonofont` de fontspec.

11. cf. 5.1.2 p. 9.

```
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = 1;
                }
            }
            if (!swapped) break;
        }
    }
\end{Piton}
```

```
void bubbleSort(int arr[], int n) {
    int temp;
    int swapped;
    for (int i = 0; i < n-1; i++) {
        swapped = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }
}
```

La commande `\PitonOptions` propose d'autres clés qui seront décrites plus loin (voir en particulier la coupure des pages et des lignes p. 9).

4.2 Les styles

4.2.1 Notion de style

L'extension piton fournit la commande `\SetPitonStyle` pour personnaliser les différents styles utilisés pour formater les éléments syntaxiques des listings Python. Ces personnalisations ont une portée qui correspond au groupe TeX courant. [12]

La commande `\SetPitonStyle` prend en argument une liste de couples *clé=valeur*. Les clés sont les noms des styles et les valeurs sont les instructions LaTeX de formatage correspondantes.

Ces instructions LaTeX doivent être des instructions de formatage du type de `\bfseries`, `\slshape`, `\color{...}`, etc. (les commandes de ce type sont parfois qualifiées de *semi-globales*). Il est aussi possible de mettre, *à la fin de la liste d'instructions*, une commande LaTeX prenant exactement un argument.

Voici un exemple qui change le style utilisé pour le nom d'une fonction Python, au moment de sa définition (c'est-à-dire après le mot-clé `def`). Elle utilise la commande `\highLight` de lua-ul (qui nécessite lui-même le chargement de luacolor).

```
\SetPitonStyle
  { Name.Function = \bfseries \hightLight[red!50] }
```

Ici, `\highLight[red!50]` doit être considéré comme le nom d'une fonction LaTeX qui prend exactement un argument, puisque, habituellement, elle est utilisée avec `\highLight[red!50]{text}`.

Avec ce réglage, on obtient : `def cube(x) : return x * x * x`

12. On rappelle que tout environnement LaTeX est, en particulier, un groupe.

L'usage des différents styles suivant le langage informatique considéré est décrit dans la partie 9, à partir de la page 26.

La commande `\PitonStyle` prend en argument le nom d'un style et permet de récupérer la valeur (en tant que liste d'instructions LaTeX) de ce style.

Par exemple, on peut écrire, dans le texte courant, `{\PitonStyle{Keyword}{function}}` et on aura le mot **function** formaté comme un mot-clé.

La syntaxe `{\PitonStyle{`*`style`*`}{...}}` est nécessaire pour pouvoir tenir compte à la fois des commandes semi-globales et des commandes à argument présentes dans la valeur courante du style *`style`*.

4.2.2 Styles locaux et globaux

Un style peut être défini de manière globale avec la commande `\SetPitonStyle`. Cela veut dire qu'il s'appliquera par défaut à tous les langages informatiques qui utilisent ce style.

Par exemple, avec la commande

```
\SetPitonStyle{Comment = \color{gray}}
```

tous les commentaires (que ce soit en Python, en C, en OCaml, etc.) seront composés en gris.

Mais il est aussi possible de définir un style localement pour un certain langage informatique en passant le nom du langage en argument optionnel (entre crochets) de la commande `\SetPitonStyle`. [13]

Par exemple, avec la commande

```
\SetPitonStyle[SQL]{Keywords = \color[HTML]{006699} \bfseries \MakeUppercase}
```

les mots-clés dans les listings SQL seront composés en lettres capitales, même s'ils s'apparaissent en minuscules dans le fichier source LaTeX (on rappelle que, en SQL, les mot-clés ne sont pas sensibles à la casse et donc forcer leur mise en capitales peut être envisagé).

Comme on s'en doute, si un langage informatique utilise un certain style et que ce style n'est pas défini localement pour ce langage, c'est la version globale qui est utilisée. Cette notion de globalité n'a pas de rapport avec la notion de liaison locale de TeX (notion de groupe TeX). [14]

Les styles fournis par défaut par piton sont tous définis globalement.

4.2.3 Le style UserFunction

Il existe un style spécial nommé `UserFunction`. Ce style s'applique aux noms des fonctions précédemment définies par l'utilisateur (par exemple, avec le langage Python, ces noms de fonctions sont ceux qui apparaissent après le mot-clé `def` dans un listing Python précédent). La valeur initiale de ce style est nulle (=vide), ce qui fait que ces noms de fonctions sont formatés comme du texte courant (en noir). Néanmoins, il est possible de changer la valeur de ce style, comme tous les autres styles, avec la commande `\SetPitonStyle`.

Dans l'exemple suivant, on règle les styles `Name.Function` et `UserFunction` de manière à que ce, quand on clique sur le nom d'une fonction Python précédemment définie par l'utilisateur, on soit renvoyé vers la définition (informatique) de cette fonction. Cette programmation utilise les fonctions `\hypertarget` et `\hyperlink` de hyperref.

```
\NewDocumentCommand{\MyDefFunction}{m}
   {\hypertarget{piton:#1}{\color[HTML]{CC00FF}{#1}}}
\NewDocumentCommand{\MyUserFunction}{m}{\hyperlink{piton:#1}{#1}}

\SetPitonStyle{Name.Function = \MyDefFunction, UserFunction = \MyUserFunction}
```

13. On rappelle que, dans piton, les noms des langages informatiques ne sont pas sensibles à la casse.
14. Du point de vue des groupes de TeX, les liaisons faites par `\SetPitonStyle` sont toujours locales.

```
def transpose(v,i,j):
    x = v[i]
    v[i] = v[j]
    v[j] = x

def passe(v):
    for in in range(0,len(v)-1):
        if v[i] > v[i+1]:
            transpose(v,i,i+1)
```

Bien sûr, la liste des noms de fonctions Python précédemment définies est gardée en mémoire de LuaLaTeX (de manière globale, c'est-à-dire indépendamment des groupes TeX). L'extension piton fournit une commande qui permet de vider cette liste : c'est la commande `\PitonClearUserFunctions`. Quand elle est utilisée sans argument, cette commande s'applique à tous les langages informatiques utilisées par l'utilisateur mais on peut spécifier en argument optionnel (entre crochets) une liste de langages informatiques auquels elle s'appliquera. [15]

4.3 Définition de nouveaux environnements

Comme l'environnement `{Piton}` a besoin d'absorber son contenu d'une manière spéciale (à peu près comme du texte verbatim), il n'est pas possible de définir de nouveaux environnements directement au-dessus de l'environnement `{Piton}` avec les commandes classiques `\newenvironment` (de LaTeX standard) et `\NewDocumentEnvironment` (de LaTeX3).
C'est pourquoi piton propose une commande `\NewPitonEnvironment`. Cette commande a la même syntaxe que la commande classique `\NewDocumentEnvironment`. [16]

Par exemple, avec l'instruction suivante, un nouvel environnement `{Python}` sera défini avec le même comportement que l'environnement `{Piton}` :

```
\NewPitonEnvironment{Python}{O{}}{\PitonOptions{#1}}{}
```

Si on souhaite un environnement `{Python}` qui compose le code inclus dans une boîte de tcolorbox, on peut écrire (à condition, bien entendu, d'avoir chargé l'extension tcolorbox) :

```
\NewPitonEnvironment{Python}{}
  {\begin{tcolorbox}}
  {\end{tcolorbox}}
```

Avec ce nouvel environnement `{Python}`, on peut écrire :

```
\begin{Python}
def carré(x):
    """Calcule le carré d'un nombre"""
    return x*x
\end{Python}
```

```
def carré(x):
    """Calcule le carré d'un nombre"""
    return x*x
```

15. On rappelle que, dans piton, les noms des langages informatiques ne sont pas sensibles à la casse.
16. Néanmoins, le spécificateur d'argument `b`, qui sert à capter le corps de l'environnement comme un argument LaTeX, n'est pas autorisé.

5 Fonctionnalités avancées

5.1 Coupure des pages et des lignes

5.1.1 Coupure des pages

Par défaut les listings produits par l'environnement `{Piton}` et par la commande `\PitonInputFile` sont insécables.
Néanmoins, la commande `\PitonOptions` propose la clé `splittable` pour autoriser de telles coupures.

— Si la clé `splittable` est utilisée sans valeur, les listings sont sécables n'importe où.

— Si la clé `splittable` est utilisée avec une valeur numérique n (qui doit être un entier naturel non nul), alors les listings seront sécables mais aucune coupure ne pourra avoir lieu entre les n premières lignes, ni entre les n dernières. De ce fait, `splittable=1` est équivalent à `splittable`.

Remarque
Même avec une couleur de fond (fixée avec `background-color`), les sauts de page sont possibles, à partir du moment où la clé `splittable` est utilisée. [17]

Nouveau 2.7 L'extension piton fournit une option `split-on-empty-lines`. Quand cette clé est active, les listings sont découpés en morceaux au niveau des lignes vides [18] du code et chaque morceau est traité par piton comme un listing autonome avant d'être envoyé à LaTeX. **De ce fait, LaTeX peut faire des sauts de page entre les morceaux**. Si la clé `splittable` est utilisée, elle reste active dans chaque morceau. Évidemment, l'utilisation conjointe de `split-on-empty-lines` et `splittable` n'a d'intérêt que si `splittable` est utilisée avec une valeur différente de 1 (par example 4), de manière à privilégier les coupures de pages au niveau des lignes vides du listing d'origine.

La clé `split-separation` permet de spécifier une liste de tokens LaTeX qui sera insérée entre les morceaux. Exemple : `\PitonOptions{split-speration = \hrule\goodbreak}`

5.1.2 Coupure des lignes

Par défaut, les éléments produits par piton ne peuvent pas être coupés par une fin de ligne. Il existe néanmoins des clés pour autoriser de telles coupures (les points de coupure possibles sont les espaces, y compris les espaces dans les chaînes Python).

— Avec la clé `break-lines-in-piton`, les coupures de ligne sont autorisées dans la commande `\piton{...}` (mais pas dans la commande `\piton|...|`, c'est-à-dire avec la syntaxe verbatim).

— Avec la clé `break-lines-in-Piton`, les coupures de ligne sont autorisées dans l'environnement `{Piton}` (d'où la lettre `P` capitale dans le nom) et les listings produits par `\PitonInputFile`.

— La clé `break-lines` est la conjonction des deux clés précédentes.

L'extension piton fournit aussi plusieurs clés pour contrôler l'apparence des coupures de ligne autorisées par `break-lines-in-Piton`.

— Avec la clé `indent-broken-lines`, l'indentation de la ligne coupée est respectée à chaque retour à la ligne.

— La clé `end-of-broken-line` correspond au symbole placé à la fin d'une ligne coupée. Sa valeur initiale est : `\hspace*{0.5em}\textbackslash`.

— La clé `continuation-symbol` correspond au symbole placé à chaque retour de ligne dans la marge gauche. Sa valeur initiale est : `+\;` (la commande `\;` insère un petit espace horizontal).

17. Avec la clé `splittable`, un environnement `{Piton}` est sécable même dans un environnement de tcolorbox (à partir du moment où la clé `breakable` de tcolorbox est utilisée). On précise cela parce que, en revanche, un environnement de tcolorbox inclus dans un autre environnement de tcolorbox n'est pas sécable, même quand les deux utilisent la clé `breakable`.

18. Les lignes considérées comme vides sont celles qui ne comportent que des espaces.

— La clé `continuation-symbol-on-indentation` correspond au symbole placé à chaque retour de ligne au niveau de l'indentation (uniquement dans le cas où la clé `indent-broken-lines` est active). Sa valeur initiale est : `$\hookrightarrow\;$`.

Le code suivant a été composé avec le réglage suivant :

```
\PitonOptions{width=12cm,break-lines,indent-broken-lines,background-color=gray!15}
```

```
def dict_of_liste(liste):
    """Convertit une liste de subrs et de descriptions de \
+   ↪ glyphes en dictionnaire"""
    dict = {}
    for liste_lettre in liste:
        if (liste_lettre[0][0:3] == 'dup'): # si c'est un subr
            nom = liste_lettre[0][4:-3]
            print("On traite le subr de numéro " + nom)
        else:
            nom = liste_lettre[0][1:-3] # si c'est un glyphe
            print("On traite le glyphe du caractère " + nom)
        dict[nom] = [traite_ligne_Postscript(k) for k in \
+       ↪ liste_lettre[1:-1]]
    return dict
```

5.2 Insertion d'une partie d'un fichier

La commande `\PitonInputFile` permet d'insérer (avec formatage) le contenu d'un fichier. En fait, il existe des mécanismes permettant de n'insérer qu'une partie du fichier en question.

— On peut spécifier la partie à insérer par les numéros de lignes (dans le fichier d'origine).

— On peut aussi spécifier la partie à insérer par des marqueurs textuels.

Dans les deux cas, si on souhaite numéroter les lignes avec les numéros des lignes du fichier d'origine, il convient d'utiliser la clé `line-numbers/absolute`.

5.2.1 Avec les numéros de lignes absolus

La commande `\PitonInputFile` propose les clés `first-line` et `last-line` pour n'insérer que la partie du fichier comprise entre les lignes correspondantes. Ne pas confondre avec la clé `line-numbers/start` qui demande un numérotage des lignes commençant à la valeur donnée à cette clé (en un sens `line-numbers/start` concerne la sortie alors que `first-line` et `last-line` concernent l'entrée).

5.2.2 Avec des marqueurs textuels

Pour utiliser cette technique, il convient d'abord de spécifier le format des marqueurs marquant le début et la fin de la partie du fichier à inclure. Cela se fait avec les deux clés `marker/beginning` et `marker/end` (usuellement dans la commande `\PitonOptions`).

Prenons d'abord un exemple.

Supposons que le fichier à inclure contienne des solutions à des exercices de programmation sur le modèle suivant :

```
#[Exercice 1] Version itérative
def fibo(n):
    if n==0: return 0
    else:
        u=0
        v=1
        for i in range(n-1):
```

```
            w = u+v
            u = v
            v = w
        return v
#<Exercice 1>
```

Les marqueurs de début de début et de fin sont les chaînes `#[Exercice 1]` et `#<Exercice 1>`. La chaîne « `Exercice 1` » sera appelée le *label* de l'exercice (ou de la partie du fichier à inclure).
Pour spécifier des marqueurs de cette sorte dans piton, on utilisera les clés `marker/beginning` et `marker/end` de la manière suivante (le caractère `#` des commentaires de Python doit être inséré sous la forme échappée `\#`).

```
\PitonOptions{ marker/beginning = \#[#1] , marker/end = \#<#1> }
```

Comme on le voit, `marker/beginning` est une expression correspondant à la fonction mathématique qui, au nom du label (par exemple `Exercice 1`), associe le marqueur de début (dans l'exemple `#[Exercice 1]`). La chaîne `#1` correspond aux occurrences de l'argument de cette fonction (c'est la syntaxe habituelle de TeX). De même pour `marker/end`.

Pour insérer une partie marquée d'un fichier, il suffit alors d'utiliser la clé `range` de `\PitonInputFile`.

```
\PitonInputFile[range = Exercice 1]{nom_du_fichier}
```

```
def fibo(n):
    if n==0: return 0
    else:
        u=0
        v=1
        for i in range(n-1):
            w = u+v
            u = v
            v = w
        return v
```

La clé `marker/include-line` demande que les lignes contenant les marqueurs soient également insérées.

```
\PitonInputFile[marker/include-lines,range = Exercice 1]{nom_du_fichier}
```

```
#[Exercice 1] Version itérative
def fibo(n):
    if n==0: return 0
    else:
        u=0
        v=1
        for i in range(n-1):
            w = u+v
            u = v
            v = w
        return v
#<Exercice 1>
```

Il existe en fait aussi les clés `begin-range` et `end-range` pour insérer plusieurs contenus marqués simultanément.
Par exemple, pour insérer les solutions des exercices 3 à 5, on pourra écrire (à condition que le fichier soit structuré correctement !) :

```
\PitonInputFile[begin-range = Exercice 3, end-range = Exercice 5]{nom_du_fichier}
```

5.3 Mise en évidence d'identificateurs

Modification 2.4

La commande `\SetPitonIdentifier` permet de changer le formatage de certains identificateurs.

Cette commande prend trois arguments : un optionnel et deux obligatoires.

— L'argument optionnel (entre crochets) indique le langage (informatique) concerné ; si cet argument est absent, les réglages faits par `\SetPitonIdentifier` s'appliqueront à tous les langages. [19]

— Le premier argument obligatoire est une liste de noms d'identificateurs séparés par des virgules.

— Le deuxième argument obligatoire est une liste d'instructions LaTeX de formatage du même type que pour les styles précédemment définis (cf. 4.2, p. 6).

Attention : Seuls les identificateurs peuvent voir leur formatage affecté. Les mots-clés et les noms de fonctions prédéfinies ne seront pas affectés, même s'ils figurent dans le premier argument de `\SetPitonIdentifier`.

```
\SetPitonIdentifier{l1,l2}{\color{red}}
\begin{Piton}
def tri(l):
    """Tri par segmentation"""
    if len(l) <= 1:
        return l
    else:
        a = l[0]
        l1 = [ x for x in l[1:] if x < a  ]
        l2 = [ x for x in l[1:] if x >= a ]
        return tri(l1) + [a] + tri(l2)
\end{Piton}
```

```
def tri(l):
    """Tri par segmentation"""
    if len(l) <= 1:
        return l
    else:
        a = l[0]
        l1 = [ x for x in l[1:] if x < a  ]
        l2 = [ x for x in l[1:] if x >= a ]
        return tri(l1) + [a] + tri(l2)
```

Avec la commande `\SetPitonIdentifiers`, on peut ajouter à un langage informatique de nouvelles fonctions prédéfinies (ou de nouveaux mots-clés, etc.) qui seront détectées par piton.

```
\SetPitonIdentifier[Python]
  {cos, sin, tan, floor, ceil, trunc, pow, exp, ln, factorial}
  {\PitonStyle{Name.Builtin}}

\begin{Piton}
from math import *
cos(pi/2)
factorial(5)
ceil(-2.3)
floor(5.4)
\end{Piton}
```

19. On rappelle que, dans piton, les noms des langages informatiques ne sont pas sensibles à la casse.

```
from math import *
cos(pi/2)
factorial(5)
ceil(-2.3)
floor(5.4)
```

5.4 Les échappements vers LaTeX

L'extension piton propose plusieurs mécanismes d'échappement vers LaTeX :

— Il est possible d'avoir des commentaires entièrement composés en LaTeX.

— Il est possible d'avoir, dans les commentaires Python, les éléments entre `$` composés en mode mathématique de LaTeX.

— Il est possible de demander à piton de détecter directement certaines commandes LaTeX avec leur argument.

— Il est possible d'insérer du code LaTeX à n'importe quel endroit d'un listing Python.

Ces mécanismes vont être détaillés dans les sous-parties suivantes.

À remarquer également que, dans le cas où piton est utilisée dans la classe beamer, piton détecte la plupart des commandes et environnements de Beamer : voir la sous-section 5.5, p. 16.

5.4.1 Les « commentaires LaTeX »

Dans ce document, on appelle « commentaire LaTeX » des commentaires qui débutent par `#>`. Tout ce qui suit ces deux caractères, et jusqu'à la fin de la ligne, sera composé comme du code LaTeX standard.
Il y a deux outils pour personnaliser ces commentaires.

— Il est possible de changer le marquage syntaxique utilisé (qui vaut initialement `#>`). Pour ce faire, il existe une clé `comment-latex`, *disponible uniquement dans le préambule du document*, qui permet de choisir les caractères qui (précédés par `#`) serviront de marqueur syntaxique.

Par exemple, avec le réglage suivant (fait dans le préambule du document) :

```
\PitonOptions{comment-latex = LaTeX}
```

les commentaires LaTeX commenceront par `#LaTeX`.

Si on donne la valeur nulle à la clé `comment-latex`, tous les commentaires Python (débutant par `#`) seront en fait des « commentaires LaTeX ».

— Il est possible de changer le formatage du commentaire LaTeX lui-même en changeant le style piton `Comment.LaTeX`.

Par exemple, avec `\SetPitonStyle{Comment.LaTeX = \normalfont\color{blue}}`, les commentaires LaTeX seront composés en bleu.

Si on souhaite qu'un croisillon (`#`) soit affiché en début de commentaire dans le PDF, on peut régler `Comment.LaTeX` de la manière suivante :

```
\SetPitonStyle{Comment.LaTeX = \color{gray}\#\normalfont\space }
```

Pour d'autres exemples de personnalisation des commentaires LaTeX, voir la partie 7.2 p. 19.

Si l'utilisateur a demandé l'affichage des numéros de ligne avec `line-numbers`, il est possible de faire référence à ce numéro de ligne avec la commande `\label` placée dans un commentaire LaTeX. [20]

20. Cette fonctionnalité est implémentée en redéfinissant, dans les environnements `{Piton}`, la commande `\label`. Il peut donc y avoir des incompatibilités avec les extensions qui redéfinissent (globalement) cette commande `\label` (comme varioref, refcheck, showlabels, etc.)

5.4.2 La clé « math-comments »

Il est possible de demander que, dans les commentaires Python normaux, c'est-à-dire débutant par `#` (et non par `#>`), les éléments placés entre symboles `$` soient composés en mode mathématique de LaTeX (le reste du commentaire restant composé en verbatim).
La clé `math-comments` (*qui ne peut être activée que dans le préambule du document*) active ce comportement.

Dans l'exemple suivant, on suppose que `\PitonOptions{math-comments}` a été utilisé dans le préambule du document.

```
\begin{Piton}
def carré(x):
    return x*x # renvoie $x^2$
\end{Piton}
```

```
def carré(x):
    return x*x # renvoie x^2
```

5.4.3 La clé « detected-commands »

La clé `detected-commands` de `\PitonOptions` permet de spécifier une liste de noms de commandes LaTeX qui seront directement détectées par piton.

— Cette clé `detected-commands` ne peut être utilisée que dans le préambule du document.

— Les noms de commandes LaTeX doivent apparaître sans la contre-oblique (ex. : `detected-commands = { emph , textbf }`).

— Ces commandes doivent être des commandes LaTeX à un seul argument obligatoire entre accolades (et ces accolades doivent être explicites).

Dans l'exemple suivant, qui est une programmation récursive de la factorielle, on décide de surligner en jaune l'appel récursif. La commande `\highLight` de lua-ul (cette extension requiert elle-même l'extension luacolor) permet de le faire avec la syntaxe `\hightLight{...}`.

On suppose que l'on a mis dans le préambule du document LaTeX l'instruction suivante :

```
\PitonOptions{detected-commands = highLight}
```

On peut alors écrire directement :

```
\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        \highLight{return n*fact(n-1)}
\end{Piton}
```

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

5.4.4 Le mécanisme « escape »

Il est aussi possible de surcharger les listings Python pour y insérer du code LaTeX à peu près n'importe où (mais entre deux lexèmes, bien entendu). Cette fonctionnalité n'est pas activée par défaut par piton. Pour l'utiliser, il faut spécifier les deux délimiteurs marquant l'échappement (le premier le commençant et le deuxième le terminant) en utilisant les clés `begin-escape` et `end-escape` (*qui ne sont accessibles que dans le préambule du document*). Les deux délimiteurs peuvent être identiques.

On reprend l'exemple précédent de la factorielle et on souhaite surligner en rose l'instruction qui contient l'appel récursif. La commande `\highLight` de lua-ul permet de le faire avec la syntaxe `\highLight{LightPink}{...}`. Du fait de la présence de l'argument optionnel entre crochets, on ne peut pas utiliser la clé `detected-commands` comme précédemment mais on peut utiliser le mécanisme « escape ».

On suppose que le préambule du document contient l'instruction :

```
\PitonOptions{begin-escape=!,end-escape=!}
```

On peut alors écrire :

```
\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        !\highLight[LightPink]{!return n*fact(n-1)!}!
\end{Piton}
```

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Attention : L'échappement vers LaTeX permis par les clés `begin-escape` et `end-escape` n'est pas actif dans les chaînes de caractères ni dans les commentaires (pour avoir un commentaire entièrement en échappement vers LaTeX, c'est-à-dire ce qui est appelé dans ce document « commentaire LaTeX », il suffit de le faire débuter par `#>`).

5.4.5 Le mécanisme « escape-math »

Le mécanisme « `escape-math` » est très similaire au mécanisme « `escape` » puisque la seule différence est que les éléments en échappement LaTeX y sont composés en mode mathématique.
On active ce mécanisme avec les clés `begin-escape-math` et `end-escape-math` (*qui ne sont accessibles que dans le préambule du document*).

Malgré la proximité technique, les usages du mécanisme « `escape-math` » sont en fait assez différents de ceux du mécanisme « `escape` ». En effet, comme le contenu en échappement est composé en mode mathématique, il est en particulier composé dans un groupe TeX et ne pourra donc pas servir à changer le formatage d'autres unités lexicales.

Dans les langages où le caractère `$` ne joue pas un rôle syntaxique important, on peut assez naturellement vouloir activer le mécanisme « `escape-math` » avec le caractère `$` :

```
\PitonOptions{begin-escape-math=$,end-escape-math=$}
```

Remarquer que le caractère `$` ne doit *pas* être protégé par une contre-oblique.

Néanmoins, il est sans doute plus prudent d'utiliser `\(` et `\)`.

```
\PitonOptions{begin-escape-math=\(,end-escape-math=\)}
```

Voici un exemple d'utilisation typique :

```
\begin{Piton}[line-numbers]
def arctan(x,n=10):
    if \(x < 0\) :
        return \(-\arctan(-x)\)
    elif \(x > 1\) :
        return \(\pi/2 - \arctan(1/x)\)
```

```
    else:
        s = \(0\)
        for \(k\) in range(\(n\)): s += \(\smash{\frac{(-1)^k}{2k+1} x^{2k+1}}\)
        return s
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0 :
        return -arctan(-x)
    elif x > 1 :
        return π/2 - arctan(1/x)
    else:
        s = 0
        for k in range(n): s += (-1)^k/(2k+1) x^(2k+1)
        return s
```

5.5 Comportement dans la classe Beamer

Première remarque
Remarquons que, comme l'environnement `{Piton}` prend son argument selon un mode verbatim, il convient, ce qui n'est pas surprenant, de l'utiliser dans des environnements `{frame}` de Beamer protégés par la clé `fragile`, c'est-à-dire débutant par `\begin{frame}[fragile]`.[21]

Quand l'extension piton est utilisée dans la classe beamer[22], le comportement de piton est légèrement modifié, comme décrit maintenant.

5.5.1 {Piton} et \PitonInputFile sont "overlay-aware"

Quand piton est utilisé avec Beamer, l'environnement `{Piton}` et la commande `\PitonInputFile` acceptent l'argument optionnel `<...>` de Beamer pour indiquer les « *overlays* » concernés.
On peut par exemple écrire :

```
\begin{Piton}<2-5>
...
\end{Piton}
```

ou aussi

```
\PitonInputFile<2-5>{mon_fichier.py}
```

5.5.2 Commandes de Beamer reconnues dans {Piton} et \PitonInputFile

Quand piton est utilisé dans la classe beamer, les commandes suivantes de beamer (classées selon leur nombre d'arguments obligatoires) sont directement reconnues dans les environnements `{Piton}` (ainsi que dans les listings composés par la commande `\PitonInputFile`, même si c'est sans doute moins utile).

— aucun argument obligatoire : `\pause`[23] ;
— un argument obligatoire : `\action`, `\alert`, `\invisible`, `\only`, `\uncover` et `\visible` ;
— deux arguments obligatoires : `\alt` ;
— trois arguments obligatoires : `\temporal`.

21. On rappelle que pour un environnement `{frame}` de Beamer qui utilise la clé `fragile`, l'instruction `\end{frame}` doit être seule sur une ligne (à l'exception d'éventuels espaces en début de ligne).
22. L'extension piton détecte la classe beamer et l'extension beamerarticle si elle est chargée précédemment, mais il est aussi possible, si le besoin s'en faisait sentir, d'activer ce comportement avec la clé `beamer` au chargement de piton : `\usepackage[beamer]{piton}`
23. On remarquera que, bien sûr, on peut aussi utiliser `\pause` dans un « commentaire LaTeX », c'est-à-dire en écrivant `#> \pause`. Ainsi, si le code Python est copié, il est interprétable par Python.

Les accolades dans les arguments obligatoires de ces commandes doivent être équilibrées (cependant, les accolades présentes dans des chaînes courtes [24] de Python ne sont pas prises en compte).

Concernant les fonctions `\alt` et `\temporal`, aucun retour à la ligne ne doit se trouver dans les arguments de ces fonctions.

Voici un exemple complet de fichier :

```
\documentclass{beamer}
\usepackage{piton}
\begin{document}
\begin{frame}[fragile]
\begin{Piton}
def string_of_list(l):
    """Convertit une liste de nombres en chaîne"""
    \only<2->{s = "{" + str(l[0])}
    \only<3->{for x in l[1:]: s = s + "," + str(x)}
    \only<4->{s = s + "}"}
    return s
\end{Piton}
\end{frame}
\end{document}
```

Dans l'exemple précédent, les accolades des deux chaînes de caractères Python `"{"` et `"}"` sont correctement interprétées (sans aucun caractère d'échappement).

5.5.3 Environnements de Beamer reconnus dans {Piton} et \PitonInputFile

Quand piton est utilisé dans la classe beamer, les environnements suivants de Beamer sont directement reconnus dans les environnements `{Piton}` (ainsi que dans les listings composés par la commande `\PitonInputFile` même si c'est sans doute moins utile) : `{actionenv}`, `{alertenv}`, `{invisibleenv}`, `{onlyenv}`, `{uncoverenv}` et `{visibleenv}`.

Il y a néanmoins une restriction : ces environnements doivent englober des *lignes entières de code Python*.

On peut par exemple écrire :

```
\documentclass{beamer}
\usepackage{piton}
\begin{document}
\begin{frame}[fragile]
\begin{Piton}
def carré(x):
    """Calcule le carré de l'argument"""
    \begin{uncoverenv}<2>
    return x*x
    \end{uncoverenv}
\end{Piton}
\end{frame}
\end{document}
```

Remarque à propos de la commande \alert et de l'environnement {alertenv} de Beamer

Beamer propose un moyen aisé de changer la couleur utilisée par l'environnement `{alertenv}` (et par la commande `\alert` qui s'appuie dessus). Par exemple, on peut écrire :

```
\setbeamercolor{alerted text}{fg=blue}
```

24. Les chaînes courtes de Python sont les chaînes (string) délimitées par les caractères ' ou " non triplés. En Python, les chaînes de caractères courtes ne peuvent pas s'étendre sur plusieurs lignes de code.

Néanmoins, dans le cas d'une utilisation à l'intérieur d'un environnement `{Piton}` un tel réglage n'est sans doute pas pertinent, puisque, justement, piton va (le plus souvent) changer la couleur des élements selon leur valeur lexicale. On préfèrera sans doute un environnement `{alertenv}` qui change la couleur de fond des éléments à mettre en évidence.

Voici un code qui effectuera ce travail en mettant un fond jaune. Ce code utilise la commande `\@highLight` de l'extension lua-ul (cette extension nécessite elle-même l'extension luacolor).

```
\setbeamercolor{alerted text}{bg=yellow!50}
\makeatletter
\AddToHook{env/Piton/begin}
  {\renewenvironment<>{alertenv}{\only#1{\@highLight[alerted text.bg]}}{}}
\makeatother
```

Ce code redéfinit localement l'environnement `{alertenv}` à l'intérieur de l'environnement `{Piton}` (on rappelle que la commande `\alert` s'appuie sur cet environnement `{alertenv}`).

5.6 Notes de pied de page dans les environnements de piton

Si vous voulez mettre des notes de pied de page dans un environnement de piton (ou bien dans un listing produit par `\PitonInputFile`, bien que cela paraisse moins pertinent dans ce cas-là) vous pouvez utiliser une paire `\footnotemark`–`\footnotetext`.

Néanmoins, il est également possible d'extraire les notes de pieds de page avec l'extension footnote ou bien l'extension footnotehyper.

Si piton est chargée avec l'option `footnote` (avec `\usepackage[footnote]{piton}`) l'extension footnote est chargée (si elle ne l'est pas déjà) et elle est utilisée pour extraire les notes de pied de page.

Si piton est chargée avec l'option `footnotehyper`, l'extension footnotehyper est chargée (si elle ne l'est pas déjà) et elle est utilisée pour extraire les notes de pied de page.

Attention : Les extensions footnote et footnotehyper sont incompatibles. L'extension footnotehyper est le successeur de l'extension footnote et devrait être utilisée préférentiellement. L'extension footnote a quelques défauts ; en particulier, elle doit être chargée après l'extension xcolor et elle n'est pas parfaitement compatible avec hyperref.

Dans ce document, l'extension piton a été chargée avec l'option `footnotehyper` et c'est pourquoi des notes peuvent être mises dans les environnements `{Piton}` : voir un exemple sur la première page de ce document.

5.7 Tabulations

Même s'il est recommandé d'indenter les listings Python avec des espaces (cf. PEP 8), piton accepte les caractères de tabulations (U+0009) en début de ligne. Chaque caractère U+0009 est remplacé par n espaces. La valeur initiale de n est 4 mais on peut la changer avec la clé `tab-size` de `\PitonOptions`.

Il existe aussi une clé `tabs-auto-gobble` qui détermine le nombre minimal de caractères U+0009 débutant chaque ligne (non vide) de l'environnement `{Piton}` et applique `gobble` avec cette valeur (avant le remplacement des caractères U+0009 par des espaces, bien entendu). Cette clé est donc similaire à la clé `auto-gobble` mais agit sur des caractères U+0009 au lieu de caractères U+0020 (espaces).

6 API pour les développeurs

La variable L3 `\l_piton_language_str` contient le nom du langage courant (en minuscules).

Nouveau 2.6

L'extension piton fournit une fonction Lua `piton.get_last_code` sans argument permettant de récupérer le code contenu dans le dernier environnement de piton.

— Les retours à la ligne (présents dans l'environnement de départ) apparaissent comme des caractères `\r` (c'est-à-dire des caractères U+000D).

— Le code fourni par `piton.get_last_code()` tient compte de l'éventuelle application d'une clé `gobble` (cf. p. 4).

— Les surcharges du code (qui entraînent des échappements vers LaTeX) ont été retirées du code fourni par `piton.get_last_code()`. Cela s'applique aux commandes LaTeX déclarées par la clé `detected-commands` (cf. partie 5.4.3) et aux éléments insérés avec le mécanisme « `escape` » (cf. partie 5.4.4).

— `piton.get_last_code` est une fonction Lua et non une chaîne de caractères : les traitements présentés précédemment sont exécutés lorsque la fonction est appelée. De ce fait, il peut être judicieux de stocker la valeur renvoyée par `piton.get_last_code()` dans une variable Lua si on doit l'utiliser plusieurs fois.

Pour un exemple d'utilisation, voir la partie concernant l'utilisation (standard) de pyluatex, partie 8.1, p. 22.

7 Exemples

7.1 Numérotation des lignes

On rappelle que l'on peut demander la numérotation des lignes des listings avec la clé `line-numbers`. Par défaut, les numéros de ligne sont composés par piton en débordement à gauche (en utilisant en interne la commande `\llap` de LaTeX).
Si on ne veut pas de débordement, on peut utiliser l'option `left-margin=auto` qui va insérer une marge adaptée aux numéros qui seront insérés (elle est plus large quand les numéros dépassent 10).

```
\PitonOptions{background-color=gray!10, left-margin = auto, line-numbers}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)        #> (appel récursif)
    elif x > 1:
        return pi/2 - arctan(1/x) #> (autre appel récursif)
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
```

```
1 def arctan(x,n=10):
2     if x < 0:
3         return -arctan(-x)         (appel récursif)
4     elif x > 1:
5         return pi/2 - arctan(1/x)  (autre appel récursif)
6     else:
7         return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

7.2 Formatage des commentaires LaTeX

On peut modifier le style `Comment.LaTeX` (avec `\SetPitonStyle`) pour faire afficher les commentaires LaTeX (qui débutent par `#>`) en butée à droite.

```
\PitonOptions{background-color=gray!10}
\SetPitonStyle{Comment.LaTeX = \hfill \normalfont\color{gray}}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)        #> appel récursif
```

```
    elif x > 1:
        return pi/2 - arctan(1/x) #> autre appel récursif
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)                                    appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x)                             autre appel récursif
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

On peut aussi faire afficher les commentaires dans une deuxième colonne à droite si on limite la largeur du code proprement dit avec la clé `width`. Dans l'exemple qui suit, on utilise la clé `width` avec la valeur spéciale `min`. Plusieurs compilations sont nécessaires.

```
\PitonOptions{width=min, background-color=gray!10}
\NewDocumentCommand{\MyLaTeXCommand}{m}{\hfill \normalfont\itshape\rlap{\quad #1}}
\SetPitonStyle{Comment.LaTeX = \MyLaTeXCommand}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)       #> appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x) #> autre appel récursif
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)                    appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x)             autre appel récursif
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

7.3 Notes dans les listings

Pour pouvoir extraire les notes (introduites par `\footnote`), l'extension `piton` doit être chargée, soit avec la clé `footnote`, soit avec la clé `footnotehyper`, comme expliqué à la section 5.6 p. 18. Dans le présent document, l'extension piton a été chargée par la clé `footnotehyper`.
Bien entendu, une commande `\footnote` ne peut apparaître que dans un commentaire LaTeX (qui débute par `#>`). Un tel commentaire peut se limiter à cette unique commande `\footnote`, comme dans l'exemple suivant.

```
\PitonOptions{background-color=gray!10}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
```

```
        return -arctan(-x)#>\footnote{Un premier appel récursif.}]
    elif x > 1:
        return pi/2 - arctan(1/x)#>\footnote{Un deuxième appel récursif.}
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x) [25]
    elif x > 1:
        return pi/2 - arctan(1/x) [26]
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

Si on utilise l'environnement {Piton} dans un environnement {minipage} de LaTeX, les notes sont, bien entendu, composées au bas de l'environnement {minipage}. Rappelons qu'une telle {minipage} ne peut être coupée par un saut de page.

```
\PitonOptions{background-color=gray!10}
\emphase\begin{minipage}{\linewidth}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)#>\footnote{Un premier appel récursif.}
    elif x > 1:
        return pi/2 - arctan(1/x)#>\footnote{Un deuxième appel récursif.}
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
\end{minipage}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x) [a]
    elif x > 1:
        return pi/2 - arctan(1/x) [b]
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

a. Un premier appel récursif.
b. Un deuxième appel récursif.

7.4 Un exemple de réglage des styles

Les styles graphiques ont été présentés à la partie 4.2, p. 6.

On présente ici un réglage de ces styles adapté pour les documents en noir et blanc. On l'utilise avec la fonte *DejaVu Sans Mono*[27] spécifiée avec la commande \setmonofont de fontspec.
Ce réglage utilise la commande \highLight de lua-ul (cette extension nécessite elle-même l'extension luacolor).

```
\setmonofont[Scale=0.85]{DejaVu Sans Mono}

\SetPitonStyle
```

25. Un premier appel récursif.
26. Un deuxième appel récursif.
27. Voir : https://dejavu-fonts.github.io

```
  {
    Number = ,
    String = \itshape ,
    String.Doc = \color{gray} \itshape ,
    Operator = ,
    Operator.Word = \bfseries ,
    Name.Builtin = ,
    Name.Function = \bfseries \highLight[gray!20] ,
    Comment = \color{gray} ,
    Comment.LaTeX = \normalfont \color{gray},
    Keyword = \bfseries ,
    Name.Namespace = ,
    Name.Class = ,
    Name.Type = ,
    InitialValues = \color{gray}
  }
```

Dans ce réglage, de nombreuses valeurs fournies aux clés sont vides, ce qui signifie que le style correspondant n'insèrera aucune instruction de formatage (l'élément sera composé dans la couleur standard, le plus souvent, en noir, etc.). Ces entrées avec valeurs nulles sont néanmoins nécessaires car la valeur initiale de ces styles dans piton n'est *pas* vide.

```
from math import pi

def arctan(x,n=10):
    """Compute the mathematical value of arctan(x)

    n is the number of terms in the sum
    """
    if x < 0:
        return -arctan(-x) # appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x)
        (on a utilisé le fait que arctan(x) + arctan(1/x) = π/2 pour x > 0)
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

8 Utilisation avec pyluatex

8.1 Utilisation standard de pyluatex

L'extension pyluatex est une extension qui permet l'exécution de code Python à partir de `lualatex` (pourvu que Python soit installé sur la machine et que la compilation soit effectuée avec `lualatex` et `--shell-escape`).
Voici, à titre d'exemple, un environnement `{PitonExecute}` qui formate un listing Python (avec piton) et qui affiche également dessous le résultat de l'exécution de ce code avec Python.

```
\NewPitonEnvironment{PitonExecute}{O{}}
  {\PitonOptions{#1}}
  {\begin{center}
   \directlua{pyluatex.execute(piton.get_last_code(), false, true, false, true)}%
   \end{center}
   \ignorespacesafterend}
```

On a utilisé la fonction Lua `piton.get_last_code` fournie dans l'API de piton : cf. partie 6, p. 18.

Cet environnement `{PitonExecute}` prend en argument optionnel (entre crochets) les options proposées par la commande `\PitonOptions`.

```
\begin{PitonExecute}[background-color=gray!15]
def carré(x):
    """Calcule le carré de l'argument"""
    return x*x
print(f'Le carré de 12 est {carré(12)}.')
\end{PitonExecute}
```

```
def carré(x):
    """Calcule le carré de l'argument"""
    return x*x
print(f'Le carré de 12 est {carré(12)}.')
```

```
Le carré de 12 est 144.
```

8.2 Utilisation de l'environnement {pythonrepl} de pyluatex

L'environnement `{pythonrepl}` de pyluatex passe son contenu à Python et renvoie ce que l'on obtient quand on fournit ce code à une boucle REPL (*read-eval-print loop*) de Python. On obtient un entrelacement d'instructions précédées par le prompt `>>>` de Python et des valeurs renvoyées par Python (et de ce qui a été demandé d'être affiché avec des `print` de Python).

Il est ensuite possible de passer cela à un environnement `{Piton}` qui va faire un coloriage syntaxique et mettre sur fond grisé les lignes correspondant aux instructions fournies à l'interpréteur Python (grâce à la clé `prompt-background-color` de `\PitonOptions`).

Voici la programmation d'un environnement `{PitonREPL}` qui effectue ce travail (pour des raisons techniques, le `!` est ici obligatoire dans la signature de l'environnement). On ne peut pas procéder comme précédemment (dans l'utilisation « standard » de pyluatex) car, bien sûr, c'est le retour fait par `{pythonrepl}` qui doit être traité par piton. De ce fait, il ne sera pas possible de mettre des surcharges (avec `detected-commands` ou le mécanisme `escape`) dans le code.

```
\ExplSyntaxOn
\NewDocumentEnvironment { PitonREPL } { ! O { } } % le ! est obligatoire
 {
   \PitonOptions
     {
       background-color=white,
       prompt-background-color = gray!10,
       #1
     }
   \PyLTVerbatimEnv
   \begin{pythonrepl}
 }
 {
   \end{pythonrepl}
   \lua_now:n
     {
       tex.print("\\begin{Piton}")
       tex.print(pyluatex.get_last_output())
       tex.print("\\end{Piton}")
       tex.print("")
     }
   \ignorespacesafterend
 }
\ExplSyntaxOff
```

Voici un exemple d'utilisation de ce nouvel environnement `{PitonREPL}`.

```
\begin{PitonREPL}
    def valeur_absolue(x):
        "Renvoie la valeur absolue de x"
        if x > 0:
           return x
        else:
           return -x

    valeur_absolue(-3)
    valeur_absolue(0)
    valeur_absolue(5)
\end{PitonREPL}
```

```
>>> def valeur_absolue(x):
...     "Renvoie la valeur absolue de x"
...     if x > 0:
...        return x
...     else:
...        return -x
...
>>> valeur_absolue(-3)
3
>>> valeur_absolue(0)
0
>>> valeur_absolue(5)
5
```

En fait, il est possible de ne pas faire afficher les prompts eux-mêmes (c'est-à-dire les chaînes de caractères `>>>` et `...`). En effet, piton propose un style pour ces éléments, qui est appelé `Prompt`. Par défaut, la valeur de ce style est vide, ce qui fait qu'aucune action n'est exécutée sur ces éléments qui sont donc affichés tels quels. En fournissant comme valeur une fonction qui se contente de gober son argument, on peut demander à ce qu'ils ne soient pas affichés.

```
\NewDocumentCommand{\Gobe}{m}{}[28]
\SetPitonStyle{ Prompt = \Gobe }
```

L'exemple précédent donne alors :

```
\begin{PitonREPL}
    def valeur_absolue(x):
        "Renvoie la valeur absolue de x"
        if x > 0:
           return x
        else:
           return -x

    valeur_absolue(-3)
    valeur_absolue(0)
    valeur_absolue(5)
\end{PitonREPL}
```

28. On a défini ici une fonction `\Gobe` mais, en fait, elle existe déjà en L3 sous le nom `\use_none:n`.

```
def valeur_absolue(x):
    "Renvoie la valeur absolue de x"
    if x > 0:
       return x
    else:
       return -x

valeur_absolue(-3)
```
```
3
```
```
valeur_absolue(0)
```
```
0
```
```
valeur_absolue(5)
```
```
5
```

9 Les styles pour les différents langages informatiques

9.1 Le langage Python

Le langage par défaut de l'extension piton est Python. Si besoin est, on peut revenir au langage Python avec `\PitonOptions{language=Python}`.

Les réglages initiaux effectués par piton dans `piton.sty` sont inspirés par le style manni de Pygments tel qu'il est appliqué au langage Python par Pygments. [29]

Style	Usage
`Number`	les nombres
`String.Short`	les chaînes de caractères courtes (entre `'` ou `"`)
`String.Long`	les chaînes de caractères longues (entre `'''` ou `"""`) sauf les chaînes de documentation (qui sont gérées par `String.Doc`)
`String`	cette clé fixe à la fois `String.Short` et `String.Long`
`String.Doc`	les chaînes de documentation (seulement entre `"""` suivant PEP 257)
`String.Interpol`	les éléments syntaxiques des champs des f-strings (c'est-à-dire les caractères `{` et `}`) ; ce style hérite des styles `String.Short` et `String.Long` (suivant la chaîne où apparaît l'interpolation)
`Interpol.Inside`	le contenu des interpolations dans les f-strings (c'est-à-dire les éléments qui se trouvent entre `{` et `}` ; si l'utilisateur n'a pas fixé ce style, ces éléments sont analysés et formatés par piton au même titre que le reste du code.
`Operator`	les opérateurs suivants : `!= == << >> - ~ + / * % = < > & . | @`
`Operator.Word`	les opérateurs suivants : `in`, `is`, `and`, `or` et `not`
`Name.Builtin`	la plupart des fonctions prédéfinies par Python
`Name.Decorator`	les décorateurs (instructions débutant par `@`)
`Name.Namespace`	le nom des modules (= bibliothèques extérieures)
`Name.Class`	le nom des classes au moment de leur définition, c'est-à-dire après le mot-clé `class`
`Name.Function`	le nom des fonctions définies par l'utilisateur *au moment de leur définition* (après le mot-clé `def`)
`UserFunction`	le nom des fonctions précédemment définies par l'utilisateur (la valeur initiale de ce paramètre est vide et ces éléments sont affichés en noir — ou plutôt dans la couleur courante)
`Exception`	les exceptions prédéfinies (ex. : `SyntaxError`)
`InitialValues`	les valeurs initiales (et le symbole `=` qui précède) des arguments optionnels dans les définitions de fonctions ; si l'utilisateur n'a pas fixé ce style, ces éléments sont analysés et formatés par piton au même titre que le reste du code.
`Comment`	les commentaires commençant par `#`
`Comment.LaTeX`	les commentaires commençant par `#>` qui sont composés par piton comme du code LaTeX (et appelés simplement « commentaires LaTeX » dans ce document)
`Keyword.Constant`	`True`, `False` et `None`
`Keyword`	les mots-clés suivants : `assert, break, case, continue, del, elif, else, except, exec, finally, for, from, global, if, import, lambda, non local, pass, raise, return, try, while, with, yield et yield from.`

29. Voir `https://pygments.org/styles/`. À remarquer que, par défaut, Pygments propose pour le style manni un fond coloré dont la couleur est la couleur HTML `#F0F3F3`. Il est possible d'avoir la même couleur dans `{Piton}` avec l'instruction : `\PitonOptions{background-color = [HTML]{F0F3F3}}`

9.2 Le langage OCaml

On peut basculer vers le langage `OCaml` avec `\PitonOptions{language = OCaml}`

On peut aussi choisir le langage `OCaml` pour un environnement `{Piton}` individuel :

```
\begin{Piton}[language=OCaml]
...
\end{Piton}
```

L'option est aussi disponible pour `\PitonInputFile` : `\PitonInputFile[language=OCaml]{...}`

Style	Usage
`Number`	les nombres
`String.Short`	les caractères (entre `'`)
`String.Long`	les chaînes de caractères, entre `"` mais aussi les *quoted-strings*
`String`	cette clé fixe à la fois `String.Short` et `String.Long`
`Operator`	les opérateurs, en particulier `+`, `-`, `/`, `*`, `@`, `!=`, `==`, `&&`
`Operator.Word`	les opérateurs suivants : `and`, `asr`, `land`, `lor`, `lsl`, `lxor`, `mod` et `or`
`Name.Builtin`	les fonctions `not`, `incr`, `decr`, `fst` et `snd`
`Name.Type`	le nom des types OCaml
`Name.Field`	le nom d'un champ de module
`Name.Constructor`	le nom des constructeurs de types (qui débutent par une majuscule)
`Name.Module`	le nom des modules
`Name.Function`	le nom des fonctions définies par l'utilisateur *au moment de leur définition* (après le mot-clé `let`)
`UserFunction`	le nom des fonctions précédemment définies par l'utilisateur (la valeur initiale de ce paramètre est vide et ces éléments sont affichés en noir — ou plutôt dans la couleur courante)
`Exception`	les exceptions prédéfinies (ex. : `End_of_File`)
`TypeParameter`	les paramétreurs de type
`Comment`	les commentaires, entre `(*` et `*)` ; ces commentaires peuvent être imbriqués
`Keyword.Constant`	`true` et `false`
`Keyword`	les mots-clés suivants : `assert`, `as`, `begin`, `class`, `constraint`, `done`, `downto`, `do`, `else`, `end`, `exception`, `external`, `for`, `function`, `functor`, `fun` , `if include`, `inherit`, `initializer`, `in` , `lazy`, `let`, `match`, `method`, `module`, `mutable`, `new`, `object`, `of`, `open`, `private`, `raise`, `rec`, `sig`, `struct`, `then`, `to`, `try`, `type`, `value`, `val`, `virtual`, `when`, `while` et `with`

9.3 Le langage C (et C++)

On peut basculer vers le langage `C` avec `\PitonOptions{language = C}`

On peut aussi choisir le langage `C` pour un environnement `{Piton}` individuel :

```
\begin{Piton}[language=C]
...
\end{Piton}
```

L'option est aussi disponible pour `\PitonInputFile` : `\PitonInputFile[language=C]{...}`

Style	Usage
`Number`	les nombres
`String.Long`	les chaînes de caractères (entre `"`)
`String.Interpol`	les éléments `%d`, `%i`, `%f`, `%c`, etc. dans les chaînes de caractères ; ce style hérite du style `String.Long`
`Operator`	les opérateurs suivants : `!= == << >> - ~ + / * % = < > & . | @`
`Name.Type`	les types prédéfinis suivants : `bool`, `char`, `char16_t`, `char32_t`, `double`, `float`, `int`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `long`, `short`, `signed`, `unsigned`, `void` et `wchar_t`
`Name.Builtin`	les fonctions prédéfinies suivantes : `printf`, `scanf`, `malloc`, `sizeof` et `alignof`
`Name.Class`	le nom des classes au moment de leur définition, c'est-à-dire après le mot-clé `class`
`Name.Function`	le nom des fonctions définies par l'utilisateur *au moment de leur définition*
`UserFunction`	le nom des fonctions précédemment définies par l'utilisateur (la valeur initiale de ce paramètre est vide et ces éléments sont affichés en noir — ou plutôt dans la couleur courante)
`Preproc`	les instructions du préprocesseur (commençant par `#`)
`Comment`	les commentaires (commençant par `//` ou entre `/*` et `*/`)
`Comment.LaTeX`	les commentaires commençant par `//>` qui sont composés par piton comme du code LaTeX (et appelés simplement « commentaires LaTeX » dans ce document)
`Keyword.Constant`	`default`, `false`, `NULL`, `nullptr` et `true`
`Keyword`	les mots-clés suivants : `alignas`, `asm`, `auto`, `break`, `case`, `catch`, `class`, `constexpr`, `const`, `continue`, `decltype`, `do`, `else`, `enum`, `extern`, `for`, `goto`, `if`, `nexcept`, `private`, `public`, `register`, `restricted`, `try`, `return`, `static`, `static_assert`, `struct`, `switch`, `thread_local`, `throw`, `typedef`, `union`, `using`, `virtual`, `volatile` et `while`

9.4 Le langage SQL

On peut basculer vers le langage `SQL` avec `\PitonOptions{language = SQL}`

On peut aussi choisir le langage `SQL` pour un environnement `{Piton}` individuel :

```
\begin{Piton}[language=SQL]
...
\end{Piton}
```

L'option est aussi disponible pour `\PitonInputFile` : `\PitonInputFile[language=SQL]{...}`

Style	Usage
`Number`	les nombres
`String.Long`	les chaînes de caractères (entre `'` et non entre `"` car les éléments entre `"` sont des noms de champs et formatés avec `Name.Field`)
`Operator`	les opérateurs suivants : `= != <> >= > < <= * + /`
`Name.Table`	les noms des tables
`Name.Field`	les noms des champs des tables
`Name.Builtin`	les fonctions prédéfinies suivantes (leur nom n'est *pas* sensible à la casse) : `avg`, `count`, `char_lenght`, `concat`, `curdate`, `current_date`, `date_format`, `day`, `lower`, `ltrim`, `max`, `min`, `month`, `now`, `rank`, `round`, `rtrim`, `substring`, `sum`, `upper` et `year`.
`Comment`	les commentaires (débutant par `--` ou bien entre `/*` et `*/`)
`Comment.LaTeX`	les commentaires commençant par `-->` qui sont composés par piton comme du code LaTeX (et appelés simplement « commentaires LaTeX » dans ce document)
`Keyword`	les mots-clés suivants (leur nom n'est *pas* sensible à la casse) : `add`, `after`, `all`, `alter`, `and`, `as`, `asc`, `between`, `by`, `change`, `column`, `create`, `cross join`, `delete`, `desc`, `distinct`, `drop`, `from`, `group`, `having`, `in`, `inner`, `insert`, `into`, `is`, `join`, `left`, `like`, `limit`, `merge`, `not`, `null`, `on`, `or`, `order`, `over`, `right`, `select`, `set`, `table`, `then`, `truncate`, `union`, `update`, `values`, `when`, `where` et `with`.

Si on souhaite que les mots-clés soient capitalisés automatiquement, on peut modifier le style `Keywords` localement pour le langage SQL avec l'instruction :

```
\SetPitonStyle[SQL]{Keywords = \bfseries \MakeUppercase}
```

9.5 Le langage « minimal »

On peut basculer vers le langage « `minimal` » avec `\PitonOptions{language = minimal}`

On peut aussi choisir le langage « `minimal` » pour un environnement `{Piton}` individuel :

```
\begin{Piton}[language=minimal]
...
\end{Piton}
```

L'option est aussi disponible pour `\PitonInputFile` : `\PitonInputFile[language=minimal]{...}`

Style	Usage
`Number`	les nombres
`String`	les chaînes de caractères (qui sont entre `"`)
`Comment`	les commentaires (qui débutent par `#`)
`Comment.LaTeX`	les commentaires commençant par `#>` qui sont composés par piton comme du code LaTeX (et appelés simplement « commentaires LaTeX » dans ce document)

Ce langage « `minimal` » est proposé par piton à l'utilisateur final pour qu'il puisse y ajouter des formatages de mots-clés avec la commande `\SetPitonIdentifier` (cf. 5.3, p. 12) et créer par exemple un langage pour pseudo-code.

Index

Autre documentation

Le document `piton.pdf` (fourni avec l'extension piton) contient une traduction anglaise de la documentation ici présente, ainsi que le code source commenté et un historique des versions.

Les versions successives du fichier `piton.sty` fournies par TeXLive sont disponibles sur le serveur SVN de TeXLive :

`https://tug.org/svn/texlive/trunk/Master/texmf-dist/tex/lualatex/piton/piton.sty`

Le développement de l'extension piton se fait sur le dépôt GitHub suivant :

`https://github.com/fpantigny/piton`

Table des matières