L'extension LaTeX piton*

F. Pantigny
fpantigny@wanadoo.fr

18 août 2025

Résumé

L'extension piton propose des outils pour composer des codes informatiques avec coloration syntaxique. Elle nécessite l'emploi de la LuaLaTeX car le travail principal est fait en utilisant la bibliothèque Lua LPEG.

1 Présentation

L'extension piton utilise la librairie Lua nommée LPEG[1] pour « parser » des listings informatiques avec 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 donc, a fortiori, elle ne requiert pas l'utilisation de `--shell-escape` lors de la compilation. 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
```

Les principaux concurrents de l'extension piton sont certainement les extensions bien connues listings et minted.

Le nom de cette extension (piton) a été choisi un peu arbitrairement en référence aux pitons d'alpinisme qui servent à gravir les montagnes.

*Ce document correspond à la version 4.8x de piton, à la date du 2025/08/09.

[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 `#>`.

L'extension piton se compose 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.

2 Utilisation de l'extension

L'extension piton doit être utilisée avec **LuaLaTeX exclusivement** : si un autre moteur de compilation (comme `latex`, `pdflatex` ou `xelatex`) est utilisé, une erreur fatale sera levée.

2.1 Choix du langage

Les langages informatiques pris en charge par piton se classent en deux catégories :

- les langages reconnus nativement par piton qui sont au nombre de cinq : Python, OCaml, SQL, C (ou plutôt C++) et deux langages minimalistes nommés `minimal`[3] et `verbatim` ;
- les langages définis par l'utilisateur avec la commande `\NewPitonLanguage` décrite p. 10 (les parseurs de ces langages ne pourront jamais être aussi précis que ceux proposés nativement par piton).

Par défaut, le langage est Python.

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

```
\PitonOptions{language = OCaml}
```

En fait, le nom des langages, pour piton, est toujours **insensible à la casse**. Dans cet exemple, on aurait tout aussi bien pu écrire `Ocaml` ou `ocaml`.

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 préférentiellement de Python mais les fonctionnalités s'appliquent aussi aux autres langages.

2.2 Chargement de l'extension

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

Elle utilise et charge l'extension xcolor. Elle n'utilise pas de programme extérieur.

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

L'extension piton fournit plusieurs outils pour composer du code informatique : 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 3.3 p. 10.

[3] Le langage `minimal` peut servir pour formater du pseudo-code : cf. p. 47.

- La commande `\PitonInputFile` doit être utilisée pour insérer et composer un fichier externe : cf. partie 5.3, p. 17.

2.4 La double syntaxe de la commande \piton

La commande `\piton` possède en fait une double syntaxe. 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` de LaTeX où l'argument est délimité entre deux caractères identiques (par ex. : `\piton|...|` ou `\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[4] de l'argument de `\piton` 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.[5]

  En revanche, comme son argument subit un développement (au sens de TeX), il faut prendre soin à ne pas utiliser dans son argument de commandes fragiles (c'est-à-dire des commandes qui ne sont ni *protected* ni *fully expandable*).

- Syntaxe `\piton|...|`

  Quand la commande `\piton` prend son argument entre deux caractères identiques (tous les caractères sont autorisés sauf `%`, `\`, `#`, `{`, `}` et l'espace), 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] Cela s'applique aux commandes commençant par une contre-oblique `\` mais également aux caractères actifs, c'est-à-dire ceux de catcode 13.

[5] La commande `\piton` peut par exemple être utilisée dans une note de bas de page. Exemple : `x = 123`.

3 Personnalisation

3.1 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.[6]
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). On peut choisir l'un des six langages prédéfinis (`Python`, `OCaml`, `C`, `SQL`, `minimal` et `verbatim`) ou bien le nom d'un langage défini par l'utilisateur avec `\NewPitonLanguage` (voir partie 4, p. 10).

  La valeur initiale est `Python`.

- La clé `font-command` contient des instructions de fonte qui seront insérées au début de chaque élément formaté par piton, que ce soit avec la commande `\piton`, l'environnement `{Piton}` ou bien la commande `\PitonInputFile` (il n'y a que les « commentaires LaTeX » pour lesquels ces instructions de fonte ne sont pas utilisées).

  La valeur initiale de ce paramètre `font-command` est `\ttfamily`, ce qui fait, que, par défaut, piton utilise la fonte mono-chasse courante.

- 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 `gobble` est utilisée sans valeur, elle se comporte comme la clé `auto-gobble`, que l'on décrit maintenant.

- 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[7] de l'environnement courant dans ce fichier. À la première utilisation du fichier par piton (au cours d'une compilation avec LaTeX), celui-ci est effacé. L'écriture du fichier ne se fait en fait qu'à la fin de la compilation avec LuaLaTeX.

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

- La clé `join` est similaire à la clé `write` mais les fichiers créés sont *joints* (comme « pièces jointes ») dans le PDF. Attention : certains lecteurs de PDF ne proposent pas d'outil permettant d'accéder à ces fichiers joints. Parmi les lecteurs qui le proposent, on peut citer le logiciel gratuit Foxit PDF Reader, disponible sur toutes les plateformes.

- La clé `print` contrôle l'affichage effectif du contenu des environnements `{Piton}` dans le PDF. Bien entendu, la valeur initiale de cette clé est `true`. Néanmoins, dans certains circonstances, il peut être utile d'utiliser `print=false` (dans le cas, par exemple, ou la clé `write`, ou bien la clé `join`, est utilisée).

[6] On rappelle que tout environnement LaTeX est, en particulier, un groupe.

[7] 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 sans les surcharges de formatage LaTeX (voir la partie 6, p. 31).

- 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`, décrite plus bas, 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`).[8]
  - La clé `line-numbers/label-empty-lines` demande que les labels (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`.[9]
  - 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. partie 5.3.2, p. 17). 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 (car on peut être amené à l'utiliser souvent).
  - 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.
  - La clé `line-numbers/format` est une liste de tokens qui est insérée avant le numéro de ligne pour le formater. Il est possible de mettre *en dernière position* de cette liste une commande LaTeX à un argument comme `\fbox`.

    La valeur initiale est `\footnotesize \color{gray}`.

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

```
\PitonOptions
  {
    line-numbers =
      {
        skip-empty-lines = false ,
        label-empty-lines = false ,
        sep = 1 em ,
        format = \footnotesize \color{blue}
      }
  }
```

- 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 7.2 p. 32.

- 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` ou la clé `max-width` décrites 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}`

[8] Avec le langage Python, les lignes vides des *docstrings* sont prises en compte.
[9] Quand la clé `split-on-empty-lines` est activée, les labels des lignes vides ne sont jamais imprimés.

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

Nouveau 4.6 Dans cette liste, la couleur spéciale `none` désigne une absence de couleur.

Exemple : `\PitonOptions{background-color = {gray!5,none}}`

- **Nouveau 4.7**

  On peut utiliser `rounded-corners` pour demander des coins arrondis pour les fonds colorés spécifiés par `background-color`. La valeur initiale de ce paramètre est 0 pt, ce qui fait que les coins ne sont pas arrondis. Si on utilise la clé `rounded-corners`, l'extension tikz doit être chargée car ces coins arrondis sont tracés avec tikz. Si tikz n'est pas chargé, une erreur sera levée à la première utilisation de la clé `rounded-corners`.

  La valeur par défaut de `rounded-corners` vaut 4 pt.[10]

- 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 7.6.2 p. 39.

- La clé `width` fixe la largeur du listing produit. La valeur initiale de ce paramètre est la valeur courante de `\linewidth`.

  Ce paramètre est utilisé pour :

  – couper les lignes trop longues (sauf, bien sûr, quand la clé `break-lines` est mise à `false` : cf. p. 19) ;

  – les fonds colorés spécifiés par les clés `background-color` et `prompt-background-color` ;

  – la largeur de la boîte LaTeX créée par la clé `box` quand celle-ci est utilisée (cf. p. 12) ;

  – la largeur de la boîte graphique créée par la clé `tcolorbox` quand celle-ci est utilisée (cf. p. 13).

- **Nouveau 4.6**

  La clé `max-width` est similaire à la clé `width` mais elle fixe la largeur maximale des lignes. Si les lignes du listings sont toutes plus courtes que la valeur fournie à `max-width`, la largeur (qui sera transmise au paramètre `width`) sera la largeur maximale des lignes du listing, c'est-à-dire la largeur naturelle du listing.

  Pour la lisibilité du code `width=min` est un raccourci pour `max-width=\linewidth`.

- En activant la clé `show-spaces-in-strings`, les espaces dans les chaînes de caractères[11] sont matérialisés par le caractère ␣ (U+2423 : OPEN BOX). Bien sûr, le caractère U+2423 doit être présent dans la fonte mono-chasse utilisée.[12]

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

- Avec la clé `show-spaces`, tous les espaces sont matérialisés par le caractère ␣ (et aucune coupure de ligne ne peut plus intervenir sur ces espaces matérialisés, même quand la clé `break-lines`[13] est active). Il faut néanmoins remarquer que les espaces en fin de ligne sont tous supprimés par piton — et ne seront donc pas représentés par ␣. Pour leur part, quand la clé `show-spaces` est active, les tabulations de début de ligne sont représentées par des flèches.

[10] Cette valeur par défaut est la valeur initiale des *rounded corners* de tikz.

[11] Pour le langage Python, cela ne s'applique que pour les chaînes courtes, c'est-à-dire celles délimitées par `'` ou `"` et, en particulier, cela ne s'applique pas pour les *doc strings*. En OCaml, cela ne s'applique pas pour les *quoted strings*.

[12] La valeur initiale de `font-command` est `\ttfamily` ce qui fait que, par défaut, l'extension piton utilise simplement la fonte mono-chasse courante.

[13] cf. 5.4.1 p. 19.

```
\begin{Piton}[language=C,line-numbers,
     splittable=4,gobble,background-color=gray!15,rounded-corners,width=min]
    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;
        }
    }
\end{Piton}
```

```c
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. 19).

3.2 Les styles

3.2.1 Notion de style

L'extension piton fournit la commande `\SetPitonStyle` pour personnaliser les différents styles utilisés pour formater les éléments syntaxiques des listings informatiques. Ces personnalisations ont une portée qui correspond au groupe TeX courant.[14]

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).

[14] On rappelle que tout environnement LaTeX est, en particulier, un groupe.

```
\SetPitonStyle
  { Name.Function = \bfseries \highLight[red!30] }
```

Ici, `\highLight[red!30]` doit être considéré comme le nom d'une fonction LaTeX qui prend exactement un argument, puisque, habituellement, elle est utilisée avec `\highLight[red!30]{`*text*`}`.

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

L'usage des différents styles suivant le langage informatique considéré est décrit dans la partie 8, à partir de la page 42.

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. Cette commande est « complètement développable » au sens de TeX.

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 potentiellement présentes dans la valeur courante du style *style*.

3.2.2 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. ou dans un langage défini avec `\NewPitonLanguage`) 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`.[15]

Par exemple, avec la commande

```
\SetPitonStyle[SQL]{Keyword = \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).[16]

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

3.2.3 La commande \rowcolor

Nouveau 4.8

L'extension piton fournit la commande `\rowcolor` qui impose un fond coloré à la ligne courante (*toute la ligne* et pas seulement la partie contenant du texte) et que l'on peut utiliser dans les styles.

La commande `\rowcolor` a une syntaxe similaire à la commande classique `\color`. Par exemple, il est possible d'écrire `\rowcolor[rgb]{0.8,1,0.8}`.

La commande `\rowcolor` est protégée contre le développement TeX.

Voici un exemple pour le langage Python où on modifie le style `String.Doc` des *documentation strings* pour avoir un fond coloré gris.

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

```
\SetPitonStyle{String.Doc = \rowcolor{gray!15}\color{black!80}}
\begin{Piton}[width=min]
def square(x):
    """Calcule le carré de x
       Deuxième ligne de la documentation"""
    return x*x
\end{Piton}
```

```
def square(x):
    """Calcule le carré de x
       Deuxième ligne de la documentation"""
    return x*x
```

3.2.4 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 `\PitonStyle{Identifier}`, ce qui fait que ces noms de fonctions sont formatés comme les autres identificateurs (c'est-à-dire, par défaut, sans formatage particulier, si ce n'est celui donné par `font-command`).
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 ce que, 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}

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)
```

(Certains lecteurs de PDF affichent un cadre autour du mot `transpose` cliquable et d'autres non.)

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 auxquels elle s'appliquera.[17]

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

3.3 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).
Avec un noyau LaTeX postérieur au 2025-06-01, il est possible d'utiliser `\NewEnvironmentCopy` sur l'environment `{Piton}` mais l'utilité est limitée.
C'est pourquoi piton propose une commande `\NewPitonEnvironment`. Cette commande a la même syntaxe que la commande classique `\NewDocumentEnvironment`.[18]

Il existe aussi les commandes suivantes similaires à celles de LaTeX3 : `\RenewPitonEnvironment`, `\DeclarePitonEnvironment` et `\ProvidePitonEnvironment`.

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 mdframed, on peut écrire :

```
\usepackage[framemethod=tikz]{mdframed} % dans le préambule

\NewPitonEnvironment{Python}{}
  {\begin{mdframed}[roundcorner=3mm]}
  {\end{mdframed}}
```

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

```
\begin{Python}
def carré(x):
    """Calcule le carré de x"""
    return x*x
\end{Python}
```

```
def carré(x):
    """Calcule le carré de x"""
    return x*x
```

On peut faire une construction similaire avec une boîte graphique de tcolorbox. Néanmoins, pour permettre une meilleure cohérence entre tcolorbox et piton, l'extension piton propose la clé `tcolorbox` : cf. p. 13.

4 Définition de nouveaux langages avec la syntaxe de listings

L'extension listings est une célèbre extension LaTeX pour formater des codes informatiques.

Elle propose une commande `\lstdefinelanguage` pour définir de nouveaux langages. Cette commande est aussi utilisée en interne par listings pour sa définition des languages (en fait, pour cela, listings utilise une commande nommée `\lst@definelanguage` mais celle-ci a la même syntaxe que `\lstdefinelanguage`).

[18] Néanmoins, le spécificateur d'argument `b`, qui sert à capter le corps de l'environnement comme un argument LaTeX, n'est pas autorisé (bien entendu).

L'extension piton propose une commande `\NewPitonLanguage` pour définir de nouveaux langages (utilisables avec les outils de piton) avec quasiment la même syntaxe que `\lstdefinelanguage`.

Précisons tout de suite que l'extension piton n'utilise *pas* cette commande pour définir les langages qu'elle propose nativement (Python, C, OCaml, SQL, `minimal` et `verbatim`), ce qui permet de proposer des parseurs plus puissants.

Par exemple, dans le fichier `lstlang1.sty`, qui est un des fichiers de definition des langages proposés par défaut par listings, on trouve les instructions suivantes (dans la version 1.10a).

```
\lstdefinelanguage{Java}%
  {morekeywords={abstract,boolean,break,byte,case,catch,char,class,%
      const,continue,default,do,double,else,extends,false,final,%
      finally,float,for,goto,if,implements,import,instanceof,int,%
      interface,label,long,native,new,null,package,private,protected,%
      public,return,short,static,super,switch,synchronized,this,throw,%
      throws,transient,true,try,void,volatile,while},%
   sensitive,%
   morecomment=[l]//,%
   morecomment=[s]{/*}{*/},%
   morestring=[b]",%
   morestring=[b]',%
  }[keywords,comments,strings]
```

Pour définir un language nommé `Java` pour piton, il suffit d'écrire le code suivant, **où le dernier argument de `\lst@definelanguage`, qui est entre crochets, a été supprimé** (en fait, les symboles `%` pourraient être supprimés sans problème).

```
\NewPitonLanguage{Java}%
  {morekeywords={abstract,boolean,break,byte,case,catch,char,class,%
      const,continue,default,do,double,else,extends,false,final,%
      finally,float,for,goto,if,implements,import,instanceof,int,%
      interface,label,long,native,new,null,package,private,protected,%
      public,return,short,static,super,switch,synchronized,this,throw,%
      throws,transient,true,try,void,volatile,while},%
   sensitive,%
   morecomment=[l]//,%
   morecomment=[s]{/*}{*/},%
   morestring=[b]",%
   morestring=[b]',%
  }
```

On peut alors utiliser le language Java comme n'importe quel autre langage prédéfini de piton.
Voici un exemple de code Java formaté dans un environnement `{Piton}` avec la clé `language=Java`.[19]

```
public class Cipher { // cryptage par le chiffre de César
    public static void main(String[] args) {
        String str = "The quick brown fox Jumped over the lazy Dog";
        System.out.println( Cipher.encode( str, 12 ));
        System.out.println( Cipher.decode( Cipher.encode( str, 12), 12 ));
    }

    public static String decode(String enc, int offset) {
        return encode(enc, 26-offset);
    }

    public static String encode(String enc, int offset) {
        offset = offset % 26 + 26;
        StringBuilder encoded = new StringBuilder();
        for (char i : enc.toCharArray()) {
```

[19] On rappelle que, pour piton, les noms de langages informatiques ne sont pas sensibles à la casse, ce qui fait que l'on aurait pu aussi bien utiliser : `language=java`.

```
            if (Character.isLetter(i)) {
                if (Character.isUpperCase(i)) {
                    encoded.append((char) ('A' + (i - 'A' + offset) % 26 ));
                } else {
                    encoded.append((char) ('a' + (i - 'a' + offset) % 26 ));
                }
            } else {
                encoded.append(i);
            }
        }
        return encoded.toString();
    }
}
```

Les clés de la commande `\lstdefinelanguage` de listings prises en charge par `\NewPitonLanguage` sont : `morekeywords`, `otherkeywords`, `sensitive`, `keywordsprefix`, `moretexcs`, `morestring` (avec les lettres `b`, `d`, `s` et `m`), `morecomment` (avec les lettres `i`, `l`, `s` et `n`), `moredelim` (avec les lettres `i`, `l`, `s`, `*` et `**`), `moredirectives`, `tag`, `alsodigit`, `alsoletter` et `alsoother`.
Pour la description de ces clés, on renvoie à la documentation de listings (taper `texdoc listings` dans un terminal).

Par exemple, pour formater du code LaTeX, on pourra créer le language suivant :

```
\NewPitonLanguage{LaTeX}{keywordsprefix = \ , alsoother = @_ }
```

Initialement, les caractères `@` et `_` sont considérés comme des lettres car de nombreux langages de programmation les autorisent dans les mots-clés et les identificateurs. Avec `alsoother = @_`, on les retire de la catégorie des lettres.

5 Fonctionnalités avancées

5.1 La clé « box »

Nouveau 4.6

Si on souhaite composer un listing dans une boîte de LaTeX, on doit utiliser la clé `box`. Cette clé prend comme valeur `c`, `t` ou `b` correspondant au paramètre de position verticale (comme dans un environnement `{minipage}` de LaTeX). La valeur par défaut est `c` (comme pour `{minipage}`).

L'emploi de la clé `box` active `width=min` (sauf, bien sûr, si on utilise explicitement `width` ou `max-width`). Pour les clés `width` et `max-width`, cf. p. 6.

```
\begin{center}
\PitonOptions{box,background-color=gray!15}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
def cube(x):
    return x*x*x
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def cube(x):
    return x*x*x
```

Il est possible d'utiliser la clé `box` avec une valeur numérique pour `width` (5 cm dans l'exemple qui suit).

```
\begin{center}
\PitonOptions{box,width=5cm,background-color=gray!15}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
def cube(x):
    return x*x*x
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def cube(x):
    return x*x*x
```

Voici un exemple avec la clé `max-width`.

```
\begin{center}
\PitonOptions{box=t,max-width=7cm,background-color=gray!15}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
\begin{Piton}
def P(x):
    return 24*x**8 - 7*x**7 + 12*x**6 -4*x**5 + 4*x**3 + x**2 - 5*x + 2
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def P(x):
    return 24*x**8 - 7*x**7 + \
12*x**6 -4*x**5 + 4*x**3 + x**2 - \
5*x + 2
```

5.2 La clé « tcolorbox »

L'extension piton propose une clé `tcolorbox` qui facilite l'utilisation de l'extension tcolorbox en coordination avec l'extension piton. Néanmoins, l'extension piton ne charge *pas* tcolorbox et l'utilisateur final doit l'avoir chargée. Il doit aussi avoir chargé la librairie `breakable` de tcolorbox avec `\tcbuselibrary{breakable}` dans le préambule du document LaTeX. Si ce n'est pas le cas, une erreur sera levée à la première utilisation de la clé `tcolorbox`.

Quand la clé `tcolorbox` est utilisée, le listing formaté par piton est inclus dans un environnement `{tcolorbox}`. Cela s'applique aussi bien à la commande `\PitonInputFile` qu'à un environnement `{Piton}` (ou, plus généralement, à un environnement défini par `\NewPitonEnvironment` : cf. p. 10). Dans le cas où la clé `splittable` de piton est utilisée (cf. p. 20), la boîte graphique créée par tcolorbox sera sécable par un saut de page.

Dans le présent document, on a, de plus, chargé, dans le préambule du document LaTeX, la librairie `skins` de tcolorbox et activé la *skin* `enhanced` pour avoir une meilleure apparence au niveau du saut de page.

```
\tcbuselibrary{skins,breakable} % dans le préambule
\tcbset{enhanced}               % dans le préambule
```

```
\begin{Piton}[tcolorbox,splittable=3]
def carré(x):
    """Calcule le carré de x"""
    return x*x
...
def carré(x):
    """Calcule le carré de x"""
    return x*x
\end{Piton}
```

```
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
```

```
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
```

Bien sûr, si on veut changer la couleur du fond, on n'utilise pas `background-color` de piton mais les outils fournis par tcolorbox (la clé `colback` pour la couleur du fond).

Si on souhaite ajuster la largeur de la boîte graphique au contenu, il suffit d'utiliser la clé `width=min` fournie par piton (cf. p. 6). On peut aussi utiliser `width` ou `max-width` avec une valeur numérique. L'environnement est sécable si la clé `splittable` est utilisée (cf. p. 20).

```
\begin{Piton}[tcolorbox,width=min,splittable=3]
def carré(x):
    """Calcule le carré de x"""
    return x*x
...
def carré(x):
    """Calcule le carré de x"""
    return x*x
\end{Piton}
```

```
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
```

```
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
```

Si on souhaite que le résultat soit produit dans une boîte LaTeX (en dépit de son nom, un environnement de tcolorbox n'est pas nécessairement une boîte LaTeX), il suffit d'utiliser, conjointement avec la clé `tcolorbox`, la clé `box` fournie par piton (cf. p. 12). Bien sûr, la boîte ainsi créée ne sera *pas* sécable par un saut de page.
On rappelle que l'emploi de la clé `box` active `width=min` (sauf si on utilise explicitement `width` ou `max-width`).

```
\begin{center}
\PitonOptions{tcolorbox,box=t}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
\begin{Piton}
def cube(x):
    """Le cube de x"""
    return x*x*x
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def cube(x):
    """Le cube de x"""
    return x*x*x
```

Pour un exemple plus sophistiqué d'utilisation de la clé `tcolorbox`, voir l'exemple fourni à la page 35.

5.3 Insertion d'un fichier

5.3.1 La commande \PitonInputFile

La commande `\PitonInputFile` permet d'insérer tout ou partie d'un fichier extérieur dont le nom est passé en argument. Il existe aussi des commandes `\PitonInputFileT`, `\PitonInputFileF` et `\PitonInputFileTF` avec des arguments correspondant aux lettres `T` et `F`, arguments qui seront exécutés dans le cas où le fichier a été trouvé (lettre `T`) ou pas (lettre `F`).

La syntaxe des chemins (absolus et relatifs) est la suivante :

- Les chemins commençant par `/` sont des chemins absolus.

  Exemple : `\PitonInputFile{/Users/joe/Documents/programme.py}`

- Les chemins ne commençant pas par `/` sont relatifs au répertoire courant.

  Exemple : `\PitonInputFile{les_listings/programme.py}`

La clé `path` de la commande `\PitonOptions` permet de spécifier une *liste* de chemins où sera recherché le fichier à inclure (dans cette liste, les chemins sont séparés par des virgules). Comme précédemment, les chemins absolus doivent débuter par une oblique `/`.

5.3.2 Insertion d'une partie 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`.

Avec les numéros de lignes absolus

La commande `\PitonInputFile` propose les clés `first-line` et `last-line` qui permettent de 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).

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`.[20]

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-lines` 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>
```

[20] Du point de vue de LaTeX, les deux fonctions passées en argument doivent être *fully expandable*.

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}
```

5.4 Coupure des pages et des lignes

5.4.1 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 qui sont dans les chaînes de caractères des langages informatiques).

- 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 valeur initiale de ce paramètre est `true` (et non `false`).
- 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 (à condition que la fonte utilisée soit une fonte mono-chasse, ce qui est le cas par défaut puisque la valeur initiale de `font-command` est `\ttfamily`).
- 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).
- 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
```

Avec la clé `break-strings-anywhere`, les chaînes de caractères pourront être coupées n'importe où (et pas seulement sur les espaces).

Avec la clé `break-numbers-anywhere`, les nombres peuvent être coupés n'importe où.

5.4.2 Coupure des pages

Par défaut, les listings produits par l'environnement `{Piton}` et par la commande `\PitonInputFile` sont insécables.
Néanmoins, piton propose les clés `splittable-on-empty-lines` et `splittable` pour autoriser de telles coupures.

- La clé `splittable-on-empty-lines` autorise les coupures sur les lignes vides du listing. Les lignes considérées comme vides sont celles qui ne comportent que des espaces (et il aurait peut-être été plus habile de parler de lignes blanches).

- La clé `splittable-on-empty-lines` peut bien sûr être insuffisante et c'est pourquoi piton propose la clé `splittable`.

  Quand la clé `splittable` est utilisée avec la valeur numérique n (qui doit être un entier naturel non nul) le listing pourra être coupé n'importe où avec cette exception qu'aucune coupure ne pourra avoir lieu entre les n premières lignes, ni entre les n dernières.[21]

  Par exemple, `splittable = 4` pourrait être un réglage raisonnable.

  Employée sans argument, la clé `splittable` est équivalente à `splittable = 1`, et les listings sont alors sécables n'importe où (ce n'est pas recommandable).

  La valeur initiale de la clé `splittable` vaut 100, ce qui fait que les listings ne sont pas sécables.

Même avec une couleur de fond (fixée avec `background-color`), les sauts de page sont possibles, à partir du moment où `splittable-on-empty-lines` ou `splittable` est utilisée.

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` de tcolorbox.

On illustre ce point avec le code suivant (l'environnement `{tcolorbox}` dans lequel nous nous trouvons utilise la clé `breakable`).

```
\begin{Piton}[background-color=gray!30,rounded-corners,width=min,splittable=4]
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
...
def carré(x):
    """Calcule le carré de x"""
    return x*x
\end{Piton}
```

```
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
```

[21] Remarquer que l'on parle des lignes du listing d'origine, une telle ligne pouvant être composée sur plusieurs lignes dans le PDF final si la clé `break-lines-in-Piton` est utilisée.

```
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
```

5.5 Découpe d'un listing en sous-listings

L'exension piton fournit la clé `split-on-empty-lines`, qui ne doit pas être confondue avec la clé `splittable-on-empty-lines` définie précédemment.

Pour comprendre le fonctionnement de la clé `split-on-empty-lines`, il faut imaginer que l'on a à composer un fichier informatique qui contient une succession de définitions de fonctions informatiques. Dans la plupart des langages informatiques, ces définitions successives sont séparées par des lignes vides (ou plutôt des lignes blanches, c'est-à-dire des lignes qui ne contiennent que des espaces).

La clé `split-on-empty-lines` coupe le listing au niveau des lignes vides. Les lignes vides successives sont supprimées et remplacées par le contenu du paramètre correspondant à la clé `split-separation`.

- Ce paramètre doit contenir du matériel à insérer en *mode vertical* de TeX. On peut, par exemple, mettre la primitive TeX `\hrule`.
- La valeur initiale de ce paramètre est `\vspace{\baselineskip}\vspace{-1.25pt}`, ce qui, au final, correspond à une ligne vide dans le PDF produit (cet espace vertical est supprimé s'il tombe au niveau d'un saut de page).

Chaque morceau du code informatique est formaté (de manière autonome) dans un environnement dont le nom est donné par la clé `env-used-by-split`. La valeur initiale de ce paramètre est, sans surprise, `Piton` et les différents morceaux sont donc composés dans des environnements `{Piton}`. Si on décide de donner une autre valeur à la clé `env-used-by-split`, on doit bien sûr donner le nom d'un environnement créé par `\NewPitonEnvironment` (cf. partie 3.3, p. 10).

Chaque morceau du listing de départ étant composé dans son environnement, il dispose de sa propre numérotation des lignes (si la clé `line-numbers` est active) et de son propre fond coloré (si la clé `background-color` est utilisée), séparé des fonds des autres morceaux. Si elle est active, la clé `splittable` s'applique de manière autonome dans chaque morceau. Bien sûr, des sauts de page peuvent intervenir entre les différents morceaux du code, quelle que soit la valeur de la clé `splittable`.

```
\begin{Piton}[split-on-empty-lines,background-color=gray!15,line-numbers]
def carré(x):
    """Calcule le carré de x"""
    return x*x

def cube(x):
    """Calcule le cube de x"""
    return x*x*x
\end{Piton}
```

```
1 def carré(x):
2     """Calcule le carré de x"""
3     return x*x
```

```
1 def cube(x):
2     """Calcule le cube de x"""
3     return x*x*x
```

Attention : Comme chaque morceau est traité de manière indépendante, les commandes spécifiées par `detected-commands` ou `raw-detected-commands` (cf. p. 25) et les commandes et environnements de Beamer automatiquement détectés par piton ne doivent pas enjamber les lignes vides du listing de départ.

5.6 Mise en évidence d'identificateurs

La commande `\SetPitonIdentifier` permet de changer automatiquement le formatage de certains identificateurs en se fondant sur leur nom.

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.[22]
- 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. 3.2, p. 7).

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`.

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

```
\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}
```

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

5.7 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, 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.

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 5.8, p. 27.

5.7.1 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 7.3 p. 33.

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.[23] De même, la commande `\zlabel` du paquetage zref peut être utilisée.[24] La clé `label-as-zlabel` qui est disponible dans `\PitonOptions` dans le préambule du document permet d'utiliser `\label` à la place de `\zlabel` dans les commentaires LaTeX (ce qui est le comportement par défaut de zref en général).

5.7.2 La clé « math-comments »

Il est possible de demander que, dans les commentaires, 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 x²
```

[23] 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.)

[24] Y compris la commande `\zcref` de zref-clever.

5.7.3 La clé « detected-commands » et ses variantes

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 apparaître explicitement dans le listing).
- Ces commandes doivent être des commandes qui s'exécutent en mode horizontal de LaTeX (à l'intérieur d'une ligne de code).
- Ces commandes doivent être **protégées**[25] contre le développement au sens de TeX (car la commande `\piton` développe son argument avant de le passer à Lua pour analyse syntaxique).

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[26] permet de le faire facilement avec la syntaxe `\highLight{...}`.

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

\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)
```

La clé `raw-detected-commands` est similaire à la clé `detected-commands` mais piton ne fera pas d'analyse syntaxique des arguments des commandes LaTeX ainsi détectées.
S'il y a un retour à la ligne dans un argument d'une commande faisant l'objet d'un `raw-detected-commands`, celui-ci sera remplacé par une espace comme le fait LaTeX par défaut.

Supposons, par exemple, que l'on souhaite, dans le texte courant d'un document traitant des bases de données, introduire des spécifications de tables SQL par le nom de la table, suivi, entre parenthèses, par les noms de ses champs (ex. : `client (non, prénom)`).
Si on insère cet élément dans une commande `\piton`, le mot *client* ne va pas être reconnu comme un nom de table mais comme un nom de champ. On peut définir une commande personnelle `\NomTable` à appliquer à la main aux noms des tables. Pour cela, on la déclare avec `raw-detected-commands` pour que son argument ne soit pas réanalysé par piton (cette réanalyse entraînerait son formatage comme un nom de champ).

Dans le préambule du document LaTeX, on insère les lignes suivantes :

```
\NewDocumentCommand{\NomTable}{m}{{\PitonStyle{Name.Table}{#1}}}
\PitonOptions{language=SQL, raw-detected-commands = NomTable}
```

Dans le corps du document, l'instruction :

[25] On rappelle que, par défaut, `\NewDocumentCommand` crée des commandes protégées au contraire de la commande historique `\newcommand` de LaTeX (et de `\def` de TeX).
[26] L'extension lua-ul requiert elle-même l'extension luacolor.

```
Exemple : \piton{\NomTable{client} (nom, prénom)}
```

donne alors le résultat suivant :

Exemple : `client (nom, prénom)`

Nouveau 4.6

La clé `vertical-detected-commands` est similaire à la clé `raw-detected-commands` mais les commandes ainsi détectées doivent être des commandes LaTeX (à un argument) qui s'exécutent en mode vertical entre les lignes du listing.

On peut par exemple demander la détection de `\newpage` par

```
\PitonOptions{vertical-detected-commands = newpage}
```

et demander dans un listing un saut de page obligatoire par `\newpage{}` :

```
\begin{Piton}
def carré(x):
    return x*x    \newpage{}
def cube(x):
    return x*x*x
\end{Piton}
```

5.7.4 Le mécanisme « escape »

Il est aussi possible de surcharger les listings informatiques 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 » qui est plus général.

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 : Le mécanisme « 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 `#>`).

5.7.5 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 `\(` et `\)`, qui sont des délimiteurs du mode mathématique proposés par LaTeX.

```
\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}
```

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

5.8 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]`.[27]

Quand l'extension piton est utilisée dans la classe beamer[28], le comportement de piton est légèrement modifié, comme décrit maintenant.

[27] 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).

[28] 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}`

5.8.1 {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}
```

5.8.2 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`[29] ;
- un argument obligatoire : `\action`, `\alert`, `\invisible`, `\only`, `\uncover` et `\visible` ;
  La clé `detected-beamer-commands` permet de rajouter à cette liste de nouveaux noms de commandes (les noms de commandes ne doivent *pas* être précédés de la contre-oblique) ;
- deux arguments obligatoires : `\alt` ;
- trois arguments obligatoires : `\temporal`.

Ces commandes doivent être utilisées précédées et suivies d'un espace. Les accolades dans les arguments obligatoires de ces commandes doivent être équilibrées (cependant, les accolades présentes dans des chaînes courtes[30] 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).

[29] 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.

[30] 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.

5.8.3 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}`.

On peut ajouter de nouveaux environnements à cette liste d'environnements reconnus avec la clé `detected-beamer-environments`.

Il y a néanmoins une restriction : ces environnements doivent englober des *lignes entières de code*. Les instructions `\begin{...}` et `\end{...}` doivent être seules sur leurs lignes.

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 suite la commande `\alert` qui s'appuie dessus). Par exemple, on peut écrire :

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

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}`).

5.9 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.

Remarque importante : Si vous utilisez Beamer, il faut savoir que Beamer a son propre système d'extraction des notes de pied de page et vous n'avez donc pas à charger piton avec la clé `footnote` ou bien la clé `footnotehyper`.

Par défaut, une commande `\footnote` ne peut apparaître que dans un « commentaire LaTeX ». Mais on peut aussi ajouter la commande `\footnote` à la liste des *detected-commands* (cf. partie 5.7.3, p. 25).

Dans ce document, l'extension piton a été chargée avec l'option `footnotehyper` et on rajouté la commande `\footnote` aux *detected-commands* avec le code suivant dans le préambule du document LaTeX :

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

```
\PitonOptions{background-color=gray!15}
\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)[31]
    elif x > 1:
        return pi/2 - arctan(1/x)[32]
    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 *pas* être coupée par un saut de page.

```
\PitonOptions{background-color=gray!15}
\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}
```

[31] Un premier appel récursif.
[32] Un deuxième appel récursif.

```
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.

5.10 Tabulations

Même s'il est sans doute recommandable d'indenter les listings informatiques avec des espaces et non des tabulations[33], 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).

La clé `env-gobble` n'est pas compatible avec les tabulations.

6 API pour les développeurs

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

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` et ses variantess (cf. partie 5.7.3) et aux éléments insérés avec le mécanisme « `escape` » (cf. partie 5.7.4).
- `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 7.6.1, p. 38.

[33] Voir, par exemple, pour le langage Python, la note PEP 8.

7 Exemples

7.1 Un exemple de réglage des styles

Les styles graphiques ont été présentés à la partie 3.2, p. 7.

On présente ici un réglage de ces styles adapté pour les documents en noir et blanc.
Ce réglage utilise la commande `\highLight` de lua-ul (cette extension nécessite elle-même l'extension luacolor).

```
\SetPitonStyle
  {
    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 $\arctan(x) + \arctan(1/x) = \pi/2$ pour $x > 0$)
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

7.2 Numérotation des lignes

On rappelle que l'on peut demander la numérotation des lignes des listings avec la clé `line-numbers` (utilisée sans valeur).
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!15, 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) )
```

7.3 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!15}
\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`.

```
\PitonOptions{width=9cm, background-color=gray!15}
\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
```

7.4 La commande \rowcolor

La commande `\rowcolor` a été décrite à la partie 3.2.3, p. 8, dans la partie sur les styles. Rappelons qu'elle impose un fond coloré à la ligne courante (*toute la ligne* et pas seulement la partie contenant du texte).

Cette commande `\rowcolor` peut-être utilisée dans un style, comme illustré à la page 8 mais on peut aussi envisager de l'utiliser directement dans un listing. Il faudra néanmoins utiliser un des mécanismes d'échappement vers LaTeX fournis par piton. Dans l'exemple suivant, on utilise la clé `raw-detected-commands` (cf. p. 25). On ne pourra pas utiliser une syntaxe comme `\rowcolor[rgb]{0.8,1,0.8}` car les « commandes détectées » sont des commandes à un seul argument mais on pourra utiliser `\rowcolor{[rgb]{0.8,1,0.8}}` (syntaxe acceptée par `\rowcolor`).

```
\PitonOptions{raw-detected-commands = rowcolor} % dans le préambule

\begin{Piton}[width=min]
def fact(n):
    if n==0:
        return 1 \rowcolor{yellow!50}
    else:
        return n*fact(n-1)
\end{Piton}
```

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

Voici maintenant le même exemple avec utilisation conjointe de la clé `background-color` (cf. p. 5).

```
\begin{Piton}[width=min,background-color=gray!15]
def fact(n):
    if n==0:
        return 1 \rowcolor{yellow!50}
    else:
        return n*fact(n-1)
\end{Piton}
```

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

Comme on le constate, une marge a été ajoutée à gauche et à droite par la clé `background-color`. Pour avoir une telle marge sans couleur de fond générale, il convient d'utiliser `background-color` avec la couleur spéciale `none`.

```
\begin{Piton}[width=min,background-color=none]
def fact(n):
    if n==0:
        return 1 \rowcolor{yellow!50}
    else:
        return n*fact(n-1)
\end{Piton}
```

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

7.5 Utilisation avec tcolorbox

La clé `tcolorbox` de piton a été présentée à la page 13.
Si on souhaite l'utiliser en paramétrant la boîte graphique créée par tcolorbox (avec les clés fournies par tcolorbox), il convient d'utiliser la commande `\tcbset` de tcolorbox. Pour limiter la portée de ces réglages, le mieux est sans doute de créer un nouvel environnement avec `\NewPitonEnvironment` (cf. p. 10). Cet environnement contiendra les personnalisations de piton (avec `\PitonOptions`) et celles de tcolorbox (avec `\tcbset`).

Voici un exemple d'un tel environnement `{Python}` avec une colonne colorée à gauche pour les numéros de ligne. Cet exemple requiert que la librairie skins de tcolorbox soit chargée dans le préambule du document LaTeX avec l'instruction `\tcbuselibrary{skins}` (de manière à pouvoir utiliser la clé `enhanced`).

```
\NewPitonEnvironment{Python}{m}
  {%
    \PitonOptions
      {
        tcolorbox,
        splittable=3,
        width=min,
        line-numbers,       % active les numéros de ligne
        line-numbers =      % personnalisation des numéros de ligne
         {
           format = \footnotesize\color{white}\sffamily ,
           sep = 2.5mm
         }
      }%
    \tcbset
      {
        enhanced,
        title=#1,
        fonttitle=\sffamily,
        left = 6mm,
        top = 0mm,
        bottom = 0mm,
        overlay=
         {%
            \begin{tcbclipinterior}%
                \fill[gray!80]
                    (frame.south west) rectangle
                    ([xshift=6mm]frame.north west);
            \end{tcbclipinterior}%
         }
      }
  }
  { }
```

Dans l'exemple d'utilisation qui suit, on a illustré le fait que l'on peut forcer un saut de page dans un tel environnement avec `\newpage{}` si on a demandé à piton de détecter la commande `\newpage` avec la clé `vertical-detected-commands` (cf. p. 25) dans le préambule du document LaTeX.
Remarquer que l'on doit bien utiliser `\newpage{}` et non `\newpage` car les commandes LaTeX détectées par piton sont censées être des commandes à un argument entre accolades.

```
\PitonOptions{vertical-detected-commands = newpage}  % dans le préambule

\begin{Python}{Mon exemple}
def carré(x):
    """Calcule le carré de x"""
```

```
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x \newpage{}
def carré(x):
    """Calcule le carré de x"""
    return x*x
...
def carré(x):
    """Calcule le carré de x"""
    return x*x
\end{Python}
```

Mon exemple

```python
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
```

```
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
def carré(x):
    """Calcule le carré de x"""
    return x*x
```

```
70 def carré(x):
71     """Calcule le carré de x"""
72     return x*x
73 def carré(x):
74     """Calcule le carré de x"""
75     return x*x
76 def carré(x):
77     """Calcule le carré de x"""
78     return x*x
79 def carré(x):
80     """Calcule le carré de x"""
81     return x*x
```

7.6 Utilisation avec pyluatex

7.6.1 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. 31.

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.
```

On peut, dans cet environnement, utiliser les mécanismes d'échappement vers LaTeX de la même manière que précédemment (cf. p. 23).

```
\usepackage{luacolor,lua-ul}                % dans le préambule
\PitonOptions{detected-commands = highLight}  % dans le préambule

\begin{PitonExecute}[background-color=gray!15]
def carré(x):
    """Calcule le carré de l'argument"""
    \highLight{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.
```

7.6.2 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 (la couleur de ce fond est réglable avec la clé `prompt-background-color` dont la valuer initiale est `gray!15`).

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`, `begin-escape`, etc.) dans le code.

```
\ExplSyntaxOn
\NewDocumentEnvironment { PitonREPL } { ! O { } } % le ! est obligatoire
 {
   \PitonOptions
     {
       background-color=none, % pour avoir des petites marges
       #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}{}[34]
\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}
```

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

valeur_absolue(-3)
3
```

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

```
valeur_absolue(0)
```
```
0
```
```
valeur_absolue(5)
```
```
5
```

8 Les styles pour les différents langages informatiques

8.1 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.[35]

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 `\PitonStyle{Identifier}`, ce qui fait que ces noms de fonctions sont affichés comme les identifiants)
`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, in, lambda, non local, pass, raise, return, try, while, with, yield et yield from.`
`Identifier`	les identificateurs.

[35] 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}}`

8.2 Le langage OCaml

On peut basculer vers le langage `OCaml` avec la clé `language` : `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 : `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 `\PitonStyle{Identifier}`, ce qui fait que ces noms de fonctions sont affichés comme les identifiants)
`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`, `done`, `downto`, `do`, `else`, `exception`, `for`, `function` , `fun`, `if`, `lazy`, `match`, `mutable`, `new`, `of`, `private`, `raise`, `then`, `to`, `try` , `virtual`, `when`, `while` et `with`
`Keyword.Governing`	les mot-clés suivants : `and`, `begin`, `class`, `constraint`, `end`, `external`, `functor`, `include`, `inherit`, `initializer`, `in`, `let`, `method`, `module`, `object`, `open`, `rec`, `sig`, `struct`, `type` et `val`.
`Identifier`	les identificateurs.

8.3 Le langage C (et C++)

On peut basculer vers le langage `C` avec la clé `language` : `language = C`

Style	Usage
`Number`	les nombres
`String.Short`	les caractères (entre `'`)
`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 `\PitonStyle{Identifier}`, ce qui fait que ces noms de fonctions sont affichés comme les identifiants)
`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`
`Identifier`	les identificateurs.

8.4 Le langage SQL

On peut basculer vers le langage SQL avec la clé `language` : `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) : `abort`, `action`, `add`, `after`, `all`, `alter`, `always`, `analyze`, `and`, `as`, `asc`, `attach`, `autoincrement`, `before`, `begin`, `between`, `by`, `cascade`, `case`, `cast`, `check`, `collate`, `column`, `commit`, `conflict`, `constraint`, `create`, `cross`, `current`, `current_date`, `current_time`, `current_timestamp`, `database`, `default`, `deferrable`, `deferred`, `delete`, `desc`, `detach`, `distinct`, `do`, `drop`, `each`, `else`, `end`, `escape`, `except`, `exclude`, `exclusive`, `exists`, `explain`, `fail`, `filter`, `first`, `following`, `for`, `foreign`, `from`, `full`, `generated`, `glob`, `group`, `groups`, `having`, `if`, `ignore`, `immediate`, `in`, `index`, `indexed`, `initially`, `inner`, `insert`, `instead`, `intersect`, `into`, `is`, `isnull`, `join`, `key`, `last`, `left`, `like`, `limit`, `match`, `materialized`, `natural`, `no`, `not`, `nothing`, `notnull`, `null`, `nulls`, `of`, `offset`, `on`, `or`, `order`, `others`, `outer`, `over`, `partition`, `plan`, `pragma`, `preceding`, `primary`, `query`, `raise`, `range`, `recursive`, `references`, `regexp`, `reindex`, `release`, `rename`, `replace`, `restrict`, `returning`, `right`, `rollback`, `row`, `rows`, `savepoint`, `select`, `set`, `table`, `temp`, `temporary`, `then`, `ties`, `to`, `transaction`, `trigger`, `unbounded`, `union`, `unique`, `update`, `using`, `vacuum`, `values`, `view`, `virtual`, `when`, `where`, `window`, `with`, `without`

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}
```

8.5 Les langages définis par la commande \NewPitonLanguage

La commande `\NewPitonLanguage`, qui permet de définir de nouveaux langages en utilisant la syntaxe de l'extension listings, a été présentée p. 10.
Tous les langages définis avec la commande `\NewPitonLanguage` partagent les mêmes styles.

Style	Usage
`Number`	les nombres
`String.Long`	les chaînes de caractères définies dans `\NewPitonLanguage` par la clé `morestring`
`Comment`	les commentaires définis dans `\NewPitonLanguage` par la clé `morecomment`
`Comment.LaTeX`	les commentaires qui sont composés par piton comme du code LaTeX (et appelés simplement « commentaires LaTeX » dans ce document)
`Keyword`	les mots-clés, définis dans `\NewPitonLanguage` par les clés `morekeywords` et `moretexcs` (et également la clé `sensitive` qui indique si les mots-clés sont sensibles à la casse)
`Directive`	les directives définies dans `\NewPitonLanguage` par la clé `moredirectives`
`Tag`	les « tags » définis par la clé `tag` (les lexèmes détectés à l'intérieur d'un tag seront aussi composés avec leur propre style)
`Identifier`	les identificateurs.

Voici une possibilité de définition d'un language HTML, obtenu par une légère adaptation de la définition faite par listings (fichier `lstlang1.sty`).

```
\NewPitonLanguage{HTML}%
  {morekeywords={A,ABBR,ACRONYM,ADDRESS,APPLET,AREA,B,BASE,BASEFONT,%
      BDO,BIG,BLOCKQUOTE,BODY,BR,BUTTON,CAPTION,CENTER,CITE,CODE,COL,%
      COLGROUP,DD,DEL,DFN,DIR,DIV,DL,DOCTYPE,DT,EM,FIELDSET,FONT,FORM,%
      FRAME,FRAMESET,HEAD,HR,H1,H2,H3,H4,H5,H6,HTML,I,IFRAME,IMG,INPUT,%
      INS,ISINDEX,KBD,LABEL,LEGEND,LH,LI,LINK,LISTING,MAP,META,MENU,%
      NOFRAMES,NOSCRIPT,OBJECT,OPTGROUP,OPTION,P,PARAM,PLAINTEXT,PRE,%
      OL,Q,S,SAMP,SCRIPT,SELECT,SMALL,SPAN,STRIKE,STRING,STRONG,STYLE,%
      SUB,SUP,TABLE,TBODY,TD,TEXTAREA,TFOOT,TH,THEAD,TITLE,TR,TT,U,UL,%
      VAR,XMP,%
      accesskey,action,align,alink,alt,archive,axis,background,bgcolor,%
      border,cellpadding,cellspacing,charset,checked,cite,class,classid,%
      code,codebase,codetype,color,cols,colspan,content,coords,data,%
      datetime,defer,disabled,dir,event,error,for,frameborder,headers,%
      height,href,hreflang,hspace,http-equiv,id,ismap,label,lang,link,%
      longdesc,marginwidth,marginheight,maxlength,media,method,multiple,%
      name,nohref,noresize,noshade,nowrap,onblur,onchange,onclick,%
      ondblclick,onfocus,onkeydown,onkeypress,onkeyup,onload,onmousedown,%
      profile,readonly,onmousemove,onmouseout,onmouseover,onmouseup,%
      onselect,onunload,rel,rev,rows,rowspan,scheme,scope,scrolling,%
      selected,shape,size,src,standby,style,tabindex,text,title,type,%
      units,usemap,valign,value,valuetype,vlink,vspace,width,xmlns},%
   tag=<>,%
   alsoletter = - ,%
   sensitive=f,%
   morestring=[d]",
  }
```

8.6 Le langage « minimal »

On peut basculer vers le langage « `minimal` » avec la clé `language` : `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)
`Identifier`	les identificateurs.

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. 5.6, p. 22) et créer par exemple un langage pour pseudo-code.

8.7 Le langage « verbatim »

On peut basculer vers le langage « `verbatim` » avec la clé `language` : `language = verbatim`

Style	Usage
rien...	

Le langage « `verbatim` » ne propose aucun style et ne fait donc aucun formatage syntaxique. On peut néanmoins y utiliser le mécanisme `detected-commands` (cf. partie 5.7.3, p. 25) ainsi que le mécanisme de détection des commandes et des environnements de Beamer.

Index

Remerciements

Remerciements à Yann Salmon et Pierre Le Scornet pour leurs nombreuses suggestions pertinentes.

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