L'extension LaTeX piton *

F. Pantigny
fpantigny@wanadoo.fr

8 octobre 2022

Résumé

L'extension piton propose des outils pour composer du code Python avec une coloration syntaxique en utilisant la bibliothèque Lua LPEG. L'extension piton nécessite l'emploi de LuaLaTeX.

1 Présentation

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

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

```
from math import pi

def arctan(x,n=10):
    """Compute the value of arctan(x)
       n is the number of terms if the sum"""
    if x < 0:
         return -arctan(-x) # recursive call
    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
```

L'extension LaTeX piton est entièrement contenue dans le fichier `piton.sty`. Ce fichier peut être placé dans le répertoire courant ou dans une arborescence `texmf`. Le mieux reste néanmoins d'installer piton avec une distribution TeX comme MiKTeX, TeX Live ou MacTeX.

*Ce document correspond à la version 0.7 de piton, à la date du 2022/10/08.

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 Utilisation de l'extension

Pour utiliser l'extension piton, l'utilisateur final a seulement à charger l'extension dans son document avec la commande classique `\usepackage` et se souvenir que la compilation doit être faite avec LuaLaTeX.

L'extension piton fournit trois outils pour composer du code Python : la commande `\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. *Attention* : Cette fonction prend son argument en mode *verbatim* (comme la commande `\verb`) et, de ce fait, cette fonction ne peut pas être utilisée à l'intérieur d'un argument d'une autre fonction (on peut néanmoins l'utiliser à l'intérieur d'un environnement).

— L'environnement `{Piton}` doit être utilisé pour composer des codes de plusieurs lignes.

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

Il est possible de composer des commentaires en LaTeX en commençant par `##` (c'est un échappement vers LaTeX). Les caractères `##` eux-mêmes ne seront pas imprimés et les espaces qui les suivent sont supprimés.

3 Personnalisation

3.1 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.

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

— 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'environnment `{Piton}` et applique `gobble` avec cette valeur de n.

— Quand la clé `env-gobble` est activée, piton applique la clé `gobble` avec une valeur de n égale au nombre d'espaces sur la ligne précédant le `\end{Piton}` (si cette ligne ne comporte que des espaces).

— Avec la clé `line-numbers`, les lignes *non vides* sont numérotées dans les environnements `{Piton}` et dans les listings produits par la commande `\PitonInputFile`.

— Avec la clé `all-line-numbers`, *toutes* les lignes sont numérotées, y compris les lignes vides.

— **Nouveau 0.7** Avec la clé `resume`, le compteur de lignes n'est pas remis à zéro comme il l'est normalement au début d'un environnement `{Piton}` ou bien au début d'un listing produit par `\PitonInputFile`.

— **Nouveau 0.7** La clé `splittable` autorise un saut de page dans les environnements `{Piton}` et les listings produits par `\PitonInputFile`.

— **Nouveau 0.7** La clé `background-color` fixe la couleur de fond des environnements `{Piton}` et des listings produits par `\PitonInputFile` (ce fond a une largeur égale à la valeur courante de `\linewidth`). Même avec une couleur de fond, les sauts de page sont possibles, à partir du moment où la clé `splittable` est utilisée. [3]

3. Un environnement `{Piton}` est sécable même dans un environnement de tcolorbox, à patir 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`.

```
\PitonOptions{line-numbers,auto-gobble,background-color = gray!15}
\begin{Piton}
    from math import pi

    def arctan(x,n=10):
        """Compute the value of arctan(x)
           n is the number of terms if the sum"""
        if x < 0:
             return -arctan(-x) # recursive call
        elif x > 1:
             return pi/2 - arctan(1/x)
             ## (on a utilisé le fait que $\arctan(x)+\arctan(1/x)=\frac{\pi}{2}$ pour $x>0$)
        else
             s = 0
             for k in range(n):
                  s += (-1)**k/(2*k+1)*x**(2*k+1)
             return s
\end{Piton}
```

```python
from math import pi

def arctan(x,n=10):
    """Compute the value of arctan(x)
       n is the number of terms if the sum"""
    if x < 0:
         return -arctan(-x) # recursive call
    elif x > 1:
         return pi/2 - arctan(1/x)
         (on a utilisé le fait que $\arctan(x)+\arctan(1/x)=\frac{\pi}{2}$ pour $x>0$)
    else
         s = 0
         for k in range(n):
              s += (-1)**k/(2*k+1)*x**(2*k+1)
         return s
```

3.2 L'option escape-inside

L'option `escape-inside` doit être utilisée au chargement de piton (c'est-à-dire dans l'instruction `\usepackage`). Pour des raisons techniques, elle ne peut pas être fixée dans `\PitonOptions`. Elle prend comme valeur deux caractères qui seront utilisés pour délimiter des parties qui seront composées en LaTeX.

Dans l'exemple suivant, on suppose que l'extension piton a été chargée de la manière suivante :

```
\usepackage[escape-inside=$$]{piton}
```

Dans le code suivant, qui est une programmation récursive de la factorielle, on décide de surligner en jaune l'instruction qui contient l'appel récursif.

```
\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        $\colorbox{yellow!50}{$return n*fact(n-1)$}$
\end{Piton}
```

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

Attention : L'échappement vers LaTeX permis par les caractères de `escape-inside` n'est pas actif dans les chaînes de caractères ni dans les commentaires (pour avoir un commentaire entièrement en échappement vers LaTeX, il suffit de le faire débuter par `##`).

3.3 Les styles

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

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 `\color{...}`, `\bseries`, `\slshape`, 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`).

```
\SetPitonStyle
  { Name.Function = \bfseries \setlength{\fboxsep}{1pt}\colorbox{yellow!50} }
```

Dans cet exemple, `\colorbox{yellow!50}` doit être considéré comme le nom d'une fonction LaTeX qui prend exactement un argument, puisque, habituellement, elle est utilisée avec la syntaxe `\colorbox{yellow!50}{text}`.

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

Les différents styles sont décrits dans la table 1.

3.4 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}`.
C'est pourquoi piton propose une commande `\NewPitonEnvironment`. Cette commande a la même syntaxe que la commande classique `\NewDocumentEnvironment`.

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

Si on souhaite un environnement `{Python}` qui compose le code inclus dans une boîte de tcolorbox, on peut écrire :

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

Table 1 – Usage des différents styles

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
`String`	cette clé fixe à la fois `String.Short` et `String.Long`
`String.Doc`	les chaînes de documentation
`String.Interpol`	les éléments syntaxiques des champs des f-strings (c'est-à-dire les caractères `{`, `}` et `:`)
`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.Function`	le nom des fonctions définies par l'utilisateur *au moment de leur définition*, c'est-à-dire après le mot-clé `def`
`Name.Decorator`	les décorateurs (instructions débutant par `@` dans les classes)
`Name.Namespace`	le nom des modules (= bibliothèques extérieures)
`Name.Class`	le nom des classes au moment de leur définition
`Exception`	le nom des exceptions prédéfinies (eg : SyntaxError)
`Comment`	les commentaires commençant par `#`
`Comment.LaTeX`	les commentaires commençant par `##` qui sont composés en LaTeX par piton (`##` est une séquence d'échappement vers LaTeX)
`Keyword.Constant`	`True`, `False` et `None`
`Keyword`	les mots-clés suivants : `assert, break, case, continue, del, elif, else, except, exec, finally, for, from, global, if, import, lambda, non local, pass, raise, return, try, while, with, yield, yield from.`