The package \textit{piton}*  
F. Pantigny  
fpantigny@wanadoo.fr  
November 29, 2022  

\section*{Abstract}

The package \textit{piton} provides tools to typeset Python listings with syntactic highlighting by using the Lua library LPEG. It requires LuaLaTeX.

\section{Presentation}

The package \textit{piton} uses the Lua library LPEG\footnote{LPEG is a pattern-matching library for Lua, written in C, based on \textit{parsing expression grammars}: \url{http://www.inf.puc-rio.br/~roberto/lpeg/}.} for parsing Python listings and typeset them with syntactic highlighting. Since it uses Lua code, it works with \texttt{luatex} only (and won’t work with the other engines: \texttt{latex}, \texttt{pdflatex} and \texttt{xelatex}). It does not use external program and the compilation does not require \texttt{--shell-escape}. The compilation is very fast since all the parsing is done by the library LPEG, written in C.

Here is an example of code typeset by \textit{piton}, with the environment \texttt{\{Piton\}}.

\begin{verbatim}
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) # recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)
        (we have used that arctan(x) + arctan(1/x) = \pi/2 for x > 0)\footnote{This LaTeX escape has been done by beginning the comment by \texttt{#>}.}
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
\end{verbatim}

The package \textit{piton} is entirely contained in the file \texttt{piton.sty}. This file may be put in the current directory or in a \texttt{texmf} tree. However, the best is to install \textit{piton} with a TeX distribution such as MiKTeX, TeX Live or MacTeX.

\footnote{This document corresponds to the version 0.99 of \textit{piton}, at the date of 2022/11/29.}
2 Use of the package

2.1 Loading the package

The package piton should be loaded with the classical command \usepackage: \usepackage{piton}.

Nevertheless, we have two remarks:

- the package piton uses the package xcolor (but piton does not load xcolor: if xcolor is not loaded before the \begin{document}, a fatal error will be raised).
- the package piton must be used with LuaLaTeX exclusively: if another LaTeX engine (latex, pdflatex, xelatex,...) is used, a fatal error will be raised.

2.2 The tools provided to the user

The package piton provides several tools to typeset Python code: the command \piton, the environment \{Piton\} and the command \PitonInputFile.

- The command \piton should be used to typeset small pieces of code inside a paragraph. For example:
  \piton{def square(x): return x*x}
  \begin{verbatim}
  def square(x):
  return x*x
  \end{verbatim}
  The syntax and particularities of the command \piton are detailed below.
- The environment \{Piton\} should be used to typeset multi-lines code. Since it takes its argument in a verbatim mode, it can’t be used within the argument of a LaTeX command. For sake of customization, it’s possible to define new environments similar to the environment \{Piton\} with the command \NewPitonEnvironment: cf. 3.3 p. 5.
- The command \PitonInputFile is used to insert and typeset a whole external file.

That command takes in as optional argument (between square brackets) two keys first-line and last-line: only the part between the corresponding lines will be inserted.

2.3 The syntax of the command \piton

In fact, the command \piton is provided with a double syntax. It may be used as a standard command of LaTeX taking its argument between curly braces (\piton{...}) but it may also be used with a syntax similar to the syntax of the command \verb, that is to say with the argument delimited by two identical characters (e.g.: \piton|...|).

**Syntax \piton{...}**

When its argument is given between curly braces, the command \piton does not take its argument in verbatim mode. In particular:

- several consecutive spaces will be replaced by only one space;
- it’s not possible to use % inside the argument;
- the braces must be appear by pairs correctly nested;
- the LaTeX commands (those beginning with a backslash \ but also the active characters) are fully expanded (but not executed).

An escaping mechanism is provided: the commands \, \, \{ and \} insert the corresponding characters \, %, { and }. The last two commands are necessary only if one need to insert braces which are not balanced.

The other characters (including #, ^, _, &, $ and @) must be inserted without backslash.

Examples:

\begin{verbatim}
\piton{MyString = '\n'} MyString = '

\piton{def even(n): return n\%2==0} def even(n): return n%2==0
\piton{c="#" # an affectation } c="#" # an affectation
\piton{MyDict = {'a': 3, 'b': 4 }} MyDict = {'a': 3, 'b': 4}
\end{verbatim}
It’s possible to use the command `\piton` in the arguments of a LaTeX command.\footnote{For example, it’s possible to use the command `\piton` in a footnote. Example: `s = 'A string'`.}

- **Syntaxe `\piton`...**

  When the argument of the command `\piton` is provided between two identical characters, that argument is taken in a *verbatim mode*. Therefore, with that syntax, the command `\piton` can’t be used within the argument of another command.

  Exemples :
  
  `\piton|MyString = '|n'| `MyString = '|n|
  
  `\piton|def even(n): return n%2==0! `def even(n): return n%2==0
  
  `\piton+c="#" # an affectation + `c="#" # an affectation
  
  `\piton?MyDict = |{'a': 3, 'b': 4}| `MyDict = |{'a': 3, 'b': 4}|

3 Customization

3.1 The command `\PitonOptions`

The command `\PitonOptions` takes in as argument a comma-separated list of *key=value* pairs. The scope of the settings done by that command is the current TeX group.\footnote{We remind that an LaTeX environment is, in particular, a TeX group.}

- The key `gobble` takes in as value a positive integer *n*: the first *n* characters are discarded (before the process of highlighting of the code) for each line of the environment `\{Piton}\`. These characters are not necessarily spaces.

- When the key `auto-gobble` is in force, the extension `piton` computes the minimal value *n* of the number of consecutive spaces beginning each (non empty) line of the environment `\{Piton}\` and applies `gobble` with that value of *n*.

- When the key `env-gobble` is in force, `piton` analyzes the last line of the environment `\{Piton}\`, that is to say the line which contains `\end{Piton}` and determines whether that line contains only spaces followed by the `\end{Piton}`. If we are in that situation, `piton` computes the number *n* of spaces on that line and applies `gobble` with that value of *n*. The name of that key comes from environment `gobble`: the effect of `gobble` is set by the position of the commands `\begin{Piton}` and `\end{Piton}` which delimit the current environment.

- With the key `line-numbers`, the *non empty* lines (and all the lines of the docstrings, even the empty ones) are numbered in the environments `\{Piton}\` and in the listings resulting from the use of `\PitonInputFile`.

- With the key `all-line-numbers`, *all* the lines are numbered, including the empty ones.

- With the key `resume` the counter of lines is not set to zero at the beginning of each environment `\{Piton\}` or use of `\PitonInputFile` as it is otherwise. That allows a numbering of the lines across several environments.

- The key `left-margin` corresponds to a margin on the left. That key may be useful in conjunction with the key `line-numbers` or the key `line-all-numbers` if one does not want the numbers in an overlapping position on the left.

  It’s possible to use the key `left-margin` with the value `auto`. With that value, if the key `line-numbers` or the key `all-line-numbers` is used, a margin will be automatically inserted to fit the numbers of lines. See an example part 5.1 on page 8.

- The key `background-color` sets the background color of the environments `\{Piton\}` and the listings produced by `\PitonInputFile` (that background has a width of `\linewidth`).
• When the key show-spaces is activated, the spaces in the short strings (that is to say those delimited by ' or ") are replaced by the character \[U+2423 : \text{open box}\]. Of course, that character U+2423 must be present in the monospaced font which is used.\(^5\)

Example: \texttt{my\_string = 'Very\_good\_answer'}

\begin{Piton}
\begin{verbatim}
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) # recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)
    # (we have used that $\arctan(x) + \arctan(1/x) = \frac{\pi}{2}$ for $x>0$)
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
\end{verbatim}
\end{Piton}

The command \texttt{\PitonOptions\{line-numbers,auto-gobble,background-color = gray!15\}} provides in fact several other keys which will be described further (see in particular the “Pages breaks and line breaks” p. 7).

### 3.2 The styles

The package \texttt{piton} provides the command \texttt{\SetPitonStyle} to customize the different styles used to format the syntactic elements of the Python listings. The customizations done by that command are limited to the current \TeX group.\(^6\)

The command \texttt{\SetPitonStyle} takes in as argument a comma-separated list of key=value pairs. The keys are names of styles and the value are \LaTeX formatting instructions.

\(^5\)The package \texttt{piton} simply uses the current monospaced font. The best way to change that font is to use the command \texttt{setmonofont} of \texttt{fontspec}.

\(^6\)We remind that an \LaTeX environment is, in particular, a \TeX group.
These LaTeX instructions must be formatting instructions such as \color{...}, \bfseries, \slshape, etc. (the commands of this kind are sometimes called semi-global commands). It’s also possible to put, at the end of the list of instructions, a LaTeX command taking exactly one argument.

Here an example which changes the style used to highlight, in the definition of a Python function, the name of the function which is defined.

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

In that example, \colorbox{yellow!50} must be considered as the name of a LaTeX command which takes in exactly one argument, since, usually, it is used with the syntax \colorbox{yellow!50}{...}.

With that setting, we will have: def cube(x) : return x * x * x

The different styles are described in the table 1. The initial settings done by piton in piton.sty are inspired by the style mann\i de Pygments.\footnote{See: https://pygments.org/styles/. Remark that, by default, Pygments provides for its style mann\i a colored background whose color is the HTML color \#F0F3F3.}

3.3 Creation of new environments

Since the environment \{Piton\} has to catch its body in a special way (more or less as verbatim text), it’s not possible to construct new environments directly over the environment \{Piton\} with the classical commands \newenvironment or \NewDocumentEnvironment. That’s why piton provides a command \NewPitonEnvironment. That command takes in three mandatory arguments. That command has the same syntax as the classical environment \NewDocumentEnvironment.

With the following instruction, a new environment \{Python\} will be constructed with the same behaviour as \{Piton\}:
\NewPitonEnvironment{Python}{}{}{}

If one wishes an environment \{Python\} with takes in as optional argument (between square brackets) the keys of the command \PitonOptions, it’s possible to program as follows:
\NewPitonEnvironment{Python}{O{}}{\PitonOptions{#1}}{}

If one wishes to format Python code in a box of tcolorbox, it’s possible to define an environment \{Python\} with the following code:
\NewPitonEnvironment{Python}{\begin{tcolorbox}}{\end{tcolorbox}}

4 Advanced features

4.1 Mechanisms to escape to LaTeX

The package piton provides several mechanisms for escaping to LaTeX:

- It’s possible to compose comments entirely in LaTeX.
- It’s possible to have the elements between $ in the comments composed in LaTeX mathematical mode.
- It’s also possible to insert LaTeX code almost everywhere in a Python listing.
4.1.1 The “LaTeX comments”

In this document, we call “LaTeX comments” the comments which begins by `#>`. The code following those characters, until the end of the line, will be composed as standard LaTeX code. There is two tools to customize those comments.

- It’s possible to change the syntatic mark (which, by default, is `#>`). For this purpose, there is a key `comment-latex` available at load-time (that is to say at the \usepackage) which allows to choice the characters which, preceded by #, will be the syntatic marker.

  For example, with the following loading:
  \begin{verbatim}
  \usepackage[comment-latex = LaTeX]{piton}
  \end{verbatim}

  the LaTeX comments will begin by `#LaTeX`.

  If the key `comment-latex` is used with the empty value, all the Python comments (which begins by #) will, in fact, be “LaTeX comments”.

- It’s possible to change the formatting of the LaTeX comment itself by changing the \texttt{piton} style \texttt{Comment.LaTeX}.

  For example, with \begin{verbatim}
  \SetPitonStyle{Comment.LaTeX = \normalfont\color{blue}}
  \end{verbatim} the LaTeX comments will be composed in blue.

  If you want to have a character # at the beginning of the LaTeX comment in the PDF, you can use set `Comment.LaTeX` as follows:

  \begin{verbatim}
  \SetPitonStyle{Comment.LaTeX = \color{gray}\#\normalfont\space }
  \end{verbatim}

  For other examples of customization of the LaTeX comments, see the part 5.2 p. 9

4.1.2 The key “math-comments”

It’s possible to request that, in the standard Python comments (that is to say those beginning by # and not `#>`), the elements between $ be composed in LaTeX mathematical mode (the other elements of the comment being composed verbatim).

That feature is activated by the key `math-comments` at load-time (that is to say with the \usepackage).

In the following example, we assume that the key `math-comments` has been used when loading \texttt{piton}.

\begin{verbatim}
\begin{Piton}
def square(x):
    return x*x # compute $x^2$
\end{Piton}
def square(x):
    return x*x # compute $x^2$
\end{verbatim}

4.1.3 The mechanism “escape-inside”

It’s also possible to overwrite the Python listings to insert LaTeX code almost everywhere (but between lexical units, of course). By default, \texttt{piton} does not fix any character for that kind of escape.

In order to use this mechanism, it’s necessary to specify two characters which will delimit the escape (one for the beginning and one for the end) by using the key `escape-inside` at load-time (that is to say a the \begin{verbatim}{begin\{document\}}).

In the following example, we assume that the extension \texttt{piton} has been loaded by the following instruction.

\begin{verbatim}
\usepackage[escape-inside=\$\$]{piton}
\end{verbatim}

In the following code, which is a recursive programmation of the mathematical factorial, we decide to highlight in yellow the instruction which contains the recursive call.
\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        $\colorbox{yellow!50}{$return n*fact(n-1)$}$
\end{Piton}

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

Caution: The escape to LaTeX allowed by the characters of \texttt{escape-inside} is not active in the strings nor in the Python comments (however, it’s possible to have a whole Python comment composed in LaTeX by beginning it with 
\texttt{#>}; such comments are merely called “LaTeX comments” in this document).

4.2 Page breaks and line breaks

4.2.1 Page breaks

By default, the listings produced by the environment \texttt{Piton} and the command \texttt{\PitonInputFile} are not breakable. However, the command \texttt{\PitonOptions} provides the key \texttt{splittable} to allow such breaks.

- If the key \texttt{splittable} is used without any value, the listings are breakable everywhere.
- If the key \texttt{splittable} is used with a numeric value \texttt{n} (which must be a non-negative integer number), the listings are breakable but no break will occur within the first \texttt{n} lines and within the last \texttt{n} lines. Therefore, \texttt{splittable=1} is equivalent to \texttt{splittable}.

Even with a background color (set by \texttt{background-color}), the pages breaks are allowed, as soon as the key \texttt{splittable} is in force.\footnote{With the key \texttt{splittable}, the environments \texttt{(Piton)} are breakable, even within a (breakable) environment of \texttt{tcolorbox}. Remind that an environment of \texttt{tcolorbox} included in another environment of \texttt{tcolorbox} is not breakable, even when both environments use the key \texttt{breakable} of \texttt{tcolorbox}.}

4.2.2 Line breaks

By default, the lines of the listings produced by \texttt{Piton} and \texttt{\PitonInputFile} are not breakable. \textbf{New 0.99} There exist several keys (available in \texttt{\PitonOptions}) to allow and control such line breaks.

- The key \texttt{break-lines} actives the lines breaks. Only the spaces (even in the strings) are allowed break points.
- With the key \texttt{indent-broken-lines}, the indentation of a broken line is respected at carriage return.
- The key \texttt{end-of-broken-line} corresponds to the symbol placed at the end of a broken line. The initial value is: \texttt{hspace*{0.5em}}\texttt{textbackslash}.
- The key \texttt{continuation-symbol} corresponds to the symbol placed at each carriage return. The initial value is: \texttt{+\textbackslash}.
- The key \texttt{continuation-symbol-on-indentation} corresponds to the symbol placed at each carriage return, on the position of the indentation (only when the key \texttt{indent-broken-line} is in force). The initial value is: \texttt{$\hookrightarrow\textbackslash$}.
The following code has been composed in a \{minipage\} of width 12 cm with the following tuning:
\PitonOptions{break-lines,indent-broken-lines,background-color=gray!15}
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) )

5.2 Formatting of the LaTeX comments

It’s possible to modify the style \SetPitonStyle in order to display the LaTeX comments (which begin with #>) aligned on the right margin.

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) )

It’s also possible to display these LaTeX comments in a kind of second column by limiting the width of the Python code by an environment \minipage of LaTeX.

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) )
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 += (-1)**k/(2*k+1)*x**(2*k+1)
        return s

5.3 Notes in the listings

In order to be able to extract the notes (which are typeset with the command \footnote), the extension piton must be loaded with the key footnote or the key footenotehyper as explained in the section 4.3 p. 8. In this document, the extension piton has been loaded with the key footenotehyper. Of course, in an environment \{Piton\}, a command \footnote may appear only within a LaTeX comment (which begins with \#>). It’s possible to have comments which contain only that command \footnote. That’s the case in the following example.

\PitonOptions{background-color=gray!10}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)#\footnote{First recursive call.}
    elif x > 1:
        return pi/2 - arctan(1/x)#\footnote{Second recursive call.}
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}

If an environment \{Piton\} is used in an environment \{minipage\} of LaTeX, the notes are composed, of course, at the foot of the environment \{minipage\}. Recall that such \{minipage\} can’t be broken by a page break.

\PitonOptions{background-color=gray!10}
\emph{\begin{minipage}{\linewidth}\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)\footnote{First recursive call.}
    elif x > 1:
        return pi/2 - arctan(1/x)\footnote{Second recursive call.}
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}\end{minipage}}
def arctan(x, n=10):
    if x < 0:
        return -arctan(-x)  #\footnote{First recursive call.}
    elif x > 1:
        return pi/2 - arctan(1/x)  #\footnote{Second recursive call.}
    else:
        return sum((-1)**k/(2*k+1)*x**(2*k+1) for k in range(n))

5.4 An example of tuning of the styles

The graphical styles have been presented in the section 3.2, p. 4.
We present now an example of tuning of these styles adapted to the documents in black and white.
We use the font DejaVu Sans Mono\footnote{See: https://dejavu-fonts.github.io} specified by the command \setmonofont of fontspec.

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

\SetPitonStyle

\footnote{First recursive call.}
\footnote{Second recursive call.}
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)
        (we have used that arctan(x) + arctan(1/x) = \pi/2 for x > 0)
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s

5.5 Use with pyluatex

The package pyluatex is an extension which allows the execution of some Python code from lualatex (provided that Python is installed on the machine and that the compilation is done with lualatex and --shell-escape). Here is, for example, an environment {PitonExecute} which formats a Python listing (with piton) but display also the output of the execution of the code with Python.

\ExplSyntaxOn
\NewDocumentEnvironment { PitonExecute } { ! O { } } {
    \PyLTVerbatimEnv
    \begin{pythonq}
    } {
    \end{pythonq}
    \directlua
    \{
    tex.print("\\PitonOptions{#1}"
    tex.print("\\begin{Piton}\n    tex.print(pyluatex.get_last_code())
    tex.print("\\end{Piton}\n    tex.print(""

12
This environment \texttt{PitonExecute} takes in as optional argument (between square brackets) the options of the command \texttt{PitonOptions}. 
Table 1: Usage of the different styles

<table>
<thead>
<tr>
<th>Style</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>the numbers</td>
</tr>
<tr>
<td>String.Short</td>
<td>the short strings (between ' or &quot;)</td>
</tr>
<tr>
<td>String.Long</td>
<td>the long strings (between ''' or &quot;&quot;&quot;&quot;) except the documentation strings</td>
</tr>
<tr>
<td>String</td>
<td>that keys sets both String.Short and String.Long</td>
</tr>
<tr>
<td>String.Doc</td>
<td>the documentation strings (only between &quot;&quot;&quot;&quot; following PEP 257)</td>
</tr>
<tr>
<td>String.Interpol</td>
<td>the syntactic elements of the fields of the f-strings (that is to say the characters { and })</td>
</tr>
<tr>
<td>Operator</td>
<td>the following operators : != == &lt;&lt; &gt;&gt; - + / * % = &lt; &gt; &amp; .</td>
</tr>
<tr>
<td>Operator.Word</td>
<td>the following operators : in, is, and, or and not</td>
</tr>
<tr>
<td>Name.Builtin</td>
<td>the predefined functions of Python</td>
</tr>
<tr>
<td>Name.Function</td>
<td>the name of the functions defined by the user, at the point of their definition (that is to say after the keyword def)</td>
</tr>
<tr>
<td>Name.Decorator</td>
<td>the decorators (instructions beginning by @)</td>
</tr>
<tr>
<td>Name.Namespace</td>
<td>the name of the modules (= external libraries)</td>
</tr>
<tr>
<td>Name.Class</td>
<td>the name of the classes at the point of their definition (that is to say after the keyword class)</td>
</tr>
<tr>
<td>Exception</td>
<td>the names of the exceptions (eg: SyntaxError)</td>
</tr>
<tr>
<td>Comment</td>
<td>the comments beginning with #</td>
</tr>
<tr>
<td>Comment.LaTeX</td>
<td>the comments beginning by #&gt;, which are composed in LaTeX by piton (and simply called “LaTeX comments” in this document)</td>
</tr>
<tr>
<td>Keyword.Constant</td>
<td>True, False and None</td>
</tr>
<tr>
<td>Keyword</td>
<td>the following keywords : as, assert, break, case, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, lambda, non local, pass, raise, return, try, while, with, yield, yield from.</td>
</tr>
</tbody>
</table>
6 Implementation

6.1 Introduction

The main job of the package piton is to take in as input a Python listing and to send back to LaTeX as output that code with interlaced LaTeX instructions of formatting.

In fact, all that job is done by a LPEG called SyntaxPython. That LPEG, when matched against the string of a Python listing, returns as capture a Lua table containing data to send to LaTeX. The only thing to do after will be to apply \texttt{tex.tprint} to each element of that table.\footnote{Recall that \texttt{tex.tprint} takes in as argument a Lua table whose first component is a “catcode table” and the second element a string. The string will be sent to LaTeX with the regime of catcodes specified by the catcode table. If no catcode table is provided, the standard catcodes of LaTeX will be used.}

Consider, for example, the following Python code:

\begin{verbatim}
def parity(x):
    return x%2
\end{verbatim}

The capture returned by the LPEG SyntaxPython against that code is the Lua table containing the following elements:

\begin{verbatim}
{ "\_\_piton_begin_line: " },
{ \PitonStyle{Keyword}{" } },
{ luatexbase.catcodetables.CatcodeTableOther, "def" },
{ "\}" },
{ luatexbase.catcodetables.CatcodeTableOther, " " },
{ \PitonStyle{Name.Function}{" } },
{ luatexbase.catcodetables.CatcodeTableOther, "parity" },
{ "\}" },
{ luatexbase.catcodetables.CatcodeTableOther, "(" },
{ luatexbase.catcodetables.CatcodeTableOther, "x" },
{ luatexbase.catcodetables.CatcodeTableOther, ")" },
{ luatexbase.catcodetables.CatcodeTableOther, ":" },
{ "\_\_piton_end_line: \_\_piton_newline: \_\_piton_begin_line: " },
{ luatexbase.catcodetables.CatcodeTableOther, "return" },
{ "\}" },
{ luatexbase.catcodetables.CatcodeTableOther, " " },
{ \PitonStyle{Operator}{" } },
{ luatexbase.catcodetables.CatcodeTableOther, ")" },
{ \PitonStyle{Number}{" } },
{ luatexbase.catcodetables.CatcodeTableOther, "2" },
{ "\}" }
{ "\_\_piton_end_line: " }
\end{verbatim}

\footnote{Each line of the Python listings will be encapsulated in a pair: \_\_begin_line: -- \_\_end_line:. The token \_\_end_line: must be explicit because it will be used as marker in order to delimit the argument of the command \_\_begin_line:. Both tokens \_\_begin_line: and \_\_end_line: will be nullified in the command piton (since there can’t be lines breaks in the argument of a command \texttt{piton}).

\footnote{The lexical elements of Python for which we have a piton style will be formatted via the use of the command \texttt{PitonStyle}. Such an element is typeset in LaTeX via the syntax \texttt{\{PitonStyle\{style\}(\ldots\})} because the instructions inside an \texttt{\PitonStyle} may be both semi-global declarations like \texttt{\bfseries} and commands with one argument like \texttt{\fbox}.}

\footnote{\texttt{\PitonStyle} is a mere number which corresponds to the “catcode table” whose all characters have the catcode “other” (which means that they will be typeset by LaTeX verbatim).}

\footnote{\texttt{luatexbase.catcodetables.CatcodeTableOther} is a mere number which corresponds to the “catcode table” whose all characters have the catcode “other” (which means that they will be typeset by LaTeX verbatim).}

\footnote{We give now the LaTeX code which is sent back by Lua to TeX (we have written on several lines for legibility but no character \texttt{\r} will be sent to LaTeX). The characters which are greyed-out are sent to LaTeX with the catcode “other” (=12). All the others characters are sent with the regime of catcodes of L3 (as set by \texttt{\ExplSyntaxOn})}

We give now the LaTeX code which is sent back by Lua to TeX (we have written on several lines for legibility but no character \texttt{\r} will be sent to LaTeX). The characters which are greyed-out are sent to LaTeX with the catcode “other” (=12). All the others characters are sent with the regime of catcodes of L3 (as set by \texttt{\ExplSyntaxOn})
6.2 The L3 part of the implementation

6.2.1 Declaration of the package

```latex
\NeedsTeXFormat{LaTeX2e}
\RequirePackage{13keys2e}
\ProvidesExplPackage{piton}{\myfiledate}{\myfileversion}{Highlight Python codes with LPEG on LuaLaTeX}
\msg_new:nnn { piton } { LuaLaTeX~mandatory }{ The~package~'piton'~must~be~used~with~LuaLaTeX.\ It~won't~be~loaded. }
\sys_if_engine_luatex:F { \msg_critical:nn { piton } { LuaLaTeX~mandatory } }
\RequirePackage { luatexbase }
\bool_new:N \c_@@_footnotehyper_bool
\bool_new:N \c_@@_footnote_bool
\bool_new:N \c_@@_escape_inside_bool
\bool_new:N \c_@@_math_comments_bool
\keys_define:nn { piton / package }{ footnote .bool_set:N = \c_@@_footnote_bool , 
footnotehyper .bool_set:N = \c_@@_footnotehyper_bool ,
escape-inside .tl_set:N = \c_@@_escape_inside_tl ,
comment-latex .value_required:n = true , 
math-comments .bool_set:N = \c_@@_math_comments_bool ,
math-comments .default:n = true ,
unknown .code:n = \msg_error:nn { piton } { unknown-key~for~package } }
\msg_new:nnn { piton } { unknown-key~for~package }{ You-have-used-the-key-''\l_keys_key_str'\'-but-the-only-keys-available-here-
are-'comment-latex',-'escape-inside',-'footnote',-'footnotehyper'\-and-
'math-comments'\.-Other-keys-are-available-in-\token_to_str:N \PitonOptions.\ That-key-will-be-ignored. }
\ProcessKeysOptions { piton / package }\begingroup
```

The boolean \c_@@_footnotehyper_bool will indicate if the option footnotehyper is used.

The boolean \c_@@_footnote_bool will indicate if the option footnote is used, but quickly, it will
also be set to \texttt{true} if the option footnotehyper is used.

The following boolean corresponds to the key math-comments (only at load-time).

We define a set of keys for the options at load-time.

We process the options provided by the user at load-time.

\msg_new:nnn { piton } { unknown-key~for~package }{ Unknown-key.\ You-have-used-the-key-''\l_keys_key_str'\'-but-the-only-keys-available-here-
are-'comment-latex',-'escape-inside',-'footnote',-'footnotehyper'\-and-
'math-comments'\.-Other-keys-are-available-in-\token_to_str:N \PitonOptions.\ That-key-will-be-ignored. }
\cs_new_protected:Npn \_\_\_set_escape_char:nn #1 #2
{ \lua_now:n { piton_begin_escape = "#1" }
  \lua_now:n { piton_end_escape = "#2" }
}
\cs_generate_variant:Nn \_\_\_set_escape_char:nn { x x }
\_\_\_set_escape_char:xx
{ \tl_head:V \c_\_\_\_escape_inside_tl }
{ \tl_tail:V \c_\_\_\_escape_inside_tl }
\endgroup
\hook_gput_code:nnn { begindocument } { . }
{ \@ifpackageloaded { xcolor }
  { }
  \msg_fatal:nn { piton } { xcolor-not-loaded }
}
\msg_new:nnn { piton } { xcolor-not-loaded }
{ xcolor-not-loaded \\ The-package-'xcolor'-is-required-by-'piton'.\\ This-error-is-fatal. }
\msg_new:nnn { piton } { footnote-with-footnotehyper-package }
{ Footnote-forbidden.\\ You-can't-use-the-option-'footnote'-because-the-package-\footnotehyper-has-already-been-loaded.-\ If-you-want,-you-can-use-the-option-'footnotehyper'-and-the-footnotes-\within-the-environments-of-piton-will-be-extracted-with-the-tools-\of-the-package-footnotehyper.\\ If-you-go-on,-the-package-footnote-won't-be-loaded. }
\msg_new:nnn { piton } { footnotehyper-with-footnote-package }
{ You-can't-use-the-option-'footnotehyper'-because-the-package-\footnote-has-already-been-loaded.-\ If-you-want,-you-can-use-the-option-'footnote'-and-the-footnotes-\within-the-environments-of-piton-will-be-extracted-with-the-tools-\of-the-package-footnote.\\ If-you-go-on,-the-package-footnotehyper-won't-be-loaded. }
\bool_if:NT \c_\_\_footnote_bool
{ The class \texttt{beamer} has its own system to extract footnotes and that's why we have nothing to do if \texttt{beamer} is used. \\ \@ifclassloaded { beamer }
  \{ \bool_set_false:N \c_\_\_footnote_bool }
  \{ \@ifpackageloaded { footnotehyper }
    \{ \_\_\_error:nn { footnote-with-footnotehyper-package } \}
    \{ \usepackage { footnote } \}
  \}
}
\bool_if:NT \c_\_\_footnotehyper_bool
{ The class \texttt{beamer} has its own system to extract footnotes and that's why we have nothing to do if \texttt{beamer} is used. \\ \@ifclassloaded { beamer }
The flag \c_@@_footnote_bool is raised and so, we will only have to test \c_@@_footnote_bool in order to know if we have to insert an environment \{savenotes\}.

### 6.2.2 Parameters and technical definitions

We will compute (with Lua) the numbers of lines of the Python code and store it in the following counter.

\int_new:N \l_@@_nb_lines_int

The same for the number of non-empty lines of the Python codes.

\int_new:N \l_@@_nb_non_empty_lines_int

The following counter will be used to count the lines during the composition. It will count all the lines, empty or not empty. It won’t be used to print the numbers of the lines.

\int_new:N \g_@@_line_int

The following token list will contains the (potential) informations to write on the aux (to be used in the next compilation).

\tl_new:N \g_@@_aux_tl

The following counter corresponds to the key splittable of \PitonOptions. If the value of \l_@@_splittable_int is equal to n, then no line break can occur within the first n lines or the last n lines of the listings.

\int_new:N \l_@@_splittable_int

An initial value of splittable equal to 100 is equivalent to say that the environments \{Piton\} are unbreakable.

\int_set:Nn \l_@@_splittable_int \{ 100 \}

The following string corresponds to the key background-color of \PitonOptions.

\str_new:N \l_@@_background_color_str

We will compute the maximal width of the lines of an environment \{Piton\} in \g_@@_width_dim. We need a global variable because when the key footnote is in force, each line when be composed in an environment \{savenotes\} and (when \slim is in force) we need to exit \g_@@_width_dim from that environment.

\dim_new:N \g_@@_width_dim

The value of that dimension as written on the aux file will be stored in \l_@@_width_on_aux_dim.

\dim_new:N \l_@@_width_on_aux_dim

We will count the envoirments \{Piton\} (and, in fact, also the commands \PitonInputFile, despite the name \g_@@_env_int).

\int_new:N \g_@@_env_int

The following booleans correspond to the keys break-lines and indent-broken-lines of \PitonOptions.

\bool_new:N \l_@@_break_lines_bool
\bool_new:N \l_@@_indent_broken_lines_bool

The following token list corresponds to the key continuation-symbol.

\tl_new:N \l_@@_continuation_symbol_tl
\tl_set:Nn \l_@@_continuation_symbol_tl \{ + \}
% The following token list corresponds to the key |continuation-symbol-on-indentation|. The name has been shorten to |csoi|.
\tl_new:N \l_@@_csoi_tl
\tl_set:Nn \l_@@_csoi_tl { $ \hookrightarrow \; $ }

The following token list corresponds to the key end-of-broken-line.
\tl_new:N \l_@@_end_of_broken_line_tl
\tl_set:Nn \l_@@_end_of_broken_line_tl { \hspace*{0.5em} \textbackslash }

The following boolean corresponds to the key slim of \PitonOptions.
\bool_new:N \l_@@_slim_bool

The following dimension corresponds to the key left-margin of \PitonOptions.
\dim_new:N \l_@@_left_margin_dim

The following boolean correspond will be set when the key left-margin=auto is used.
\bool_new:N \l_@@_left_margin_auto_bool

The tabulators will be replaced by the content of the following token list.
\tl_new:N \l_@@_tab_tl
\cs_new_protected:Npn \@@_set_tab_tl:n #1
\prg_replicate:nn { #1 }{ \tl_put_right:Nn \l_@@_tab_tl { ~ } }
\@@_set_tab_tl:n { 4 }

The following integer corresponds to the key gobble.
\int_new:N \l_@@_gobble_int
\tl_new:N \l_@@_space_tl
\tl_set:Nn \l_@@_space_tl { ~ }

At each line, the following counter will count the spaces at the beginning.
\int_new:N \g_@@_indentation_int
\cs_new_protected:Npn \@@_an_indentation_space:
\int_gincr:N \g_@@_indentation_int

6.2.3 Treatment of a line of code

In the contents provided by Lua, each line of the Python code will be surrounded by \@@_begin_line: and \@@_end_line:.
\cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
\int_gzero:N \g_@@_indentation_int

Be careful: there is curryfication in the following lines.
\bool_if:NTF \l_@@_slim_bool
\{ \hcoffin_set:Nn \l_tmpa_coffin \}
\str_if_empty:NTF \l_@@_background_color_str
\{ \vcoffin_set:Nnn \l_tmpa_coffin \dim_eval:n { \linewidth - \l_@@_left_margin_dim } \}
\{ \}
\{
If the key \texttt{break-lines} is in force, we replace all the characters U+0032 (that is to say the spaces) by \texttt{\@@_breakable_space}. Remark that, except the spaces inserted in the LaTeX \texttt{comments} (and maybe in the math comments), all these spaces are of \texttt{catcode} "other" (=12) and are unbreakable.

We compute in \texttt{\g_@@_width_dim} the maximal width of the lines of the environment.

```latex
\def\@@_newline{
\int_gincr:N \g_@@_line_int
\}
\cs_new_protected:Npn \@@_newline:
{
\int_gincr:N \g_@@_line_int
\}
\cs_new_protected:Npn \@@_breakable_space:
{
\makebox[0pt][l]{\hphantom{\ttfamily { }}</\ttfamily { }}\ttfamily { }\@@_newline:
\}
```

```latex
\bool_if:NT \l_@@_break_lines_bool
{
\regex_replace_all:nnN
{x20}
{\c{\@@_breakable_space:}}
\l_tmpa_tl
\l_tmpa_tl \strut \hfil
}\hbox_set:Nn \l_tmpa_box
{
\skip_horizontal:N \l_@@_left_margin_dim
\bool_if:NT \l_@@_line_numbers_bool
{
\bool_if:NF \l_@@_all_line_numbers_bool
{\tl_if_empty:nF { #1 }\@@_print_number:
\str_if_empty:NF \l_@@_background_color_str
{\skip_horizontal:n { 0.5 em }}
\coffin_typeset:Nnnnn \l_tmpa_coffin T l \c_zero_dim \c_zero_dim
}
\dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_width_dim
{\dim_gset:Nn \g_@@_width_dim { \box_wd:N \l_tmpa_box }}
\box_set_dp:Nn \l_tmpa_box { \box_dp:N \l_tmpa_box + 1.25 pt }
\box_set_ht:Nn \l_tmpa_box { \box_ht:N \l_tmpa_box + 1.25 pt }
\tl_if_empty:NTF \l_@@_background_color_str
{\box_use_drop:N \l_tmpa_box }
{\vbox_top:n
{\hbox:n}
{\exp_args:NV \color \l_@@_background_color_str
\vrule height \box_ht:N \l_tmpa_box
depth \box_dp:N \l_tmpa_box
width \l_@@_width_on_aux_dim}
\skip_vertical:n { - \box_ht_plus_dp:N \l_tmpa_box }
\box_set wd:Nn \l_tmpa_box \l_@@_width_on_aux_dim
\box_use drop:N \l_tmpa_box }
\vspace { - 2.5 pt }
}
```
\int_compare:nNnT \g_@@_line_int > \{ \l_@@_splittable_int - 1 \}
{
\int_compare:nNnT
{ \l_@@_nb_lines_int - \g_@@_line_int } > \l_@@_splittable_int
{
\egroup
\bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
\newline
\bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
\vtop \bgroup
\}
\}

\cs_set_protected:Npn \@@_breakable_space:
{
\discretionary
{ \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
{
\hbox_overlap_left:n

\normalfont \footnotesize \color { gray }
\l_@@_continuation_symbol_tl
\skip_horizontal:n { 0.3 em }
\str_if_empty:NF \l_@@_background_color_str
{ \skip_horizontal:n { 0.5 em } }
\bool_if:NT \l_@@_indent_broken_lines_bool
{ \hbox:n
{ \prg_replicate:nn { \g_@@_indentation_int } { ~ }
{ \color { gray } \l_@@_csoi_tl }
}
}
{ \hbox { ~ } }
}

6.2.4 PitonOptions

The following parameters correspond to the keys line-numbers and all-line-numbers.
\bool_new:N \l_@@_line_numbers_bool
\bool_new:N \l_@@_all_line_numbers_bool

The following flag corresponds to the key resume.
\bool_new:N \l_@@_resume_bool

Be careful! The name of the following set of keys must be considered as public! Hence, it should not be changed.
\keys_define:nn { PitonOptions }
{ gobble .int_set:N = \l_@@_gobble_int , gobble .value_required:n = true , auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } , auto-gobble .value_forbidden:n = true , env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,}
env-gobble .value_forbidden:n = true,
line-numbers .bool_set:N = \l_@@_line_numbers_bool,
line-numbers .default:n = true,
all-line-numbers .value_forbidden:n = true,
\bool_set_true:N \l_@@_line_numbers_bool
\bool_set_true:N \l_@@_all_line_numbers_bool,
all-line-numbers .value_forbidden:n = true,
resume .bool_set:N = \l_@@_resume_bool,
resume .value_forbidden:n = true,
splittable .int_set:N = \l_@@_splittable_int,
splittable .default:n = 1,
background-color .str_set:N = \l_@@_background_color_str,
background-color .value_required:n = true,
slim .bool_set:N = \l_@@_slim_bool,
slim .default:n = true,
left-margin .code:n = \str_if_eq:nnTF { #1 } { auto }
{ \dim_zero:N \l_@@_left_margin_dim
\bool_set_true:N \l_@@_left_margin_auto_bool
}
{ \dim_set:Nn \l_@@_left_margin_dim { #1 } },
left-margin .value_required:n = true,
tab-size .code:n = \@@_set_tab_tl:n { #1 },
tab-size .value_required:n = true,
show-spaces .code:n = \tl_set:Nn \l_@@_space_tl { ␣ } , % U+2423
show-spaces .value_required:n = true,
brack-lines .bool_set:N = \l_@@_break_lines_bool,
brack-lines .default:n = true,
indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool,
indent-broken-lines .default:n = true,
end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl,
end-of-broken-line .value_required:n = true,
continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl,
continuation-symbol .value_required:n = true,
continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl,
continuation-symbol-on-indentation .value_required:n = true,
unknown .code:n = \msg_error:nnnn { piton } { Unknown~key~for~PitonOptions }
{ Unknown-key-for-PitonOptions }
{ Unknown-key. \}
The-key-\l_keys_key_str'-is-unknown-for-\token_to_str:N \PitonOptions.-
It-will-be-ignored.\}
For-a-list-of-the-available-keys,-type-H<return>.
{ The-available-keys-are-(in-alphabetic-order):
all-line-numbers,-
auto-gobble,-
brack-lines,-
continuation-symbol,-
continuation-symbol-on-indentation,-
end-of-broken-line,-
env-gobble,-gobble,-
left-margin,-
indent-broken-lines,-
line-numbers,-
resume,-
show-spaces,-
slim,}
The argument of `\PitonOptions` is provided by curryfication.

\NewDocumentCommand \PitonOptions { } { \keys_set:nn { PitonOptions } }

\subsection{The numbers of the lines}

The following counter will be used to count the lines in the code when the user requires the numbers of the lines to be printed (with `line-numbers` or `all-line-numbers`).

\int_new:N \g_@@_visual_line_int
\cs_new_protected:Npn \@@_print_number:nnn { \int_gincr:N \g_@@_visual_line_int \hbox_overlap_left:n { \color { gray } \footnotesize \int_to_arabic:n \g_@@_visual_line_int \skip_horizontal:n { 0.4 em } } }

\subsection{The command to write on the aux file}

\cs_new_protected:Npn \@@_write_aux:n { \tl_if_empty:NF \g_@@_aux_tl { \iow_now:Nn \@mainaux { \ExplSyntaxOn } \iow_now:Nx \@mainaux { \tl_gset:cn { c_@@_ \int_use:N \g_@@_env_int _ tl } { \exp_not:V \g_@@_aux_tl } } \iow_now:Nn \@mainaux { \ExplSyntaxOff } \tl_gclear:N \g_@@_aux_tl } \cs_new_protected:Npn \@@_width_to_aux:n { \bool_if:NT \l_@@_slim_bool { \str_if_empty:NF \l_@@_background_color_str { \tl_gput_right:Nx \g_@@_aux_tl { \dim_set:Nn \l_@@_width_on_aux_dim { \dim_eval:n { \g_@@_width_dim + 0.5 em } } } } } }

\subsection{The main commands and environments for the final user}

\NewDocumentCommand { \piton } { } { \peek_meaning:NTF \bgroup \@@_piton_standard \@@_piton_verbatim }
The following command is not a user command. It will be used when you will have to “rescan” some chunks of Python code. For example, if will be the initial value of the Piton style \texttt{InitialValues} (the default values of the arguments of a Python function).

Despite its name, \texttt{@@_pre_env}: will be used both in \texttt{PitonInputFile} dans in the environments such as \texttt{Piton}.
We count with Lua the number of lines of the argument. The result will be stored by Lua in \l_@@_nb_lines_int. That information will be used to allow or disallow page breaks.

\lua_now:n { piton.CountLinesFile(token.scan_argument()) } { #2 }

% If the final user has used both |left-margin=auto| and |line-numbers| or % |all-line-numbers|, we have to compute the width of the maximal number of % lines at the end of the composition of the listing to fix the correct value to % |left-margin|.
% \begin{macrocode}
\bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
{ \hbox_set:Nn \l_tmpa_box
  { \footnotesize \bool_if:NTF \l_@@_all_line_numbers_bool
    { \int_to_arabic:n { \g_@@_visual_line_int + \l_@@_nb_lines_int }
      }
    { \lua_now:n { piton.CountNonEmptyLinesFile(token.scan_argument()) } }
    \int_to_arabic:n { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
  }
\dim_set:Nn \l_@@_left_margin_dim { \box_wd:N \l_tmpa_box + 0.5em }
}
\NewDocumentCommand { \NewPitonEnvironment } { m m m m }
{ \dim_zero:N \parindent
  \ttfamily
  \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
vtop \bgroup
  \lua_now:e { piton.ParseFile(token.scan_argument(),
    \int_use:N \l_@@_first_line_int ,
    \int_use:N \l_@@_last_line_int )}
  { \end { savenotes } }
\egroup
\@@_width_to_aux:
\@@_write_aux:
}

\NewDocumentCommand { \NewPitonEnvironment } { m m m m }
{ \dim_zero:N \parindent
  \ttfamily
  \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
We count with Lua the number of lines of the argument. The result will be stored by Lua in \l_@@_nb_lines_int. That information will be used to allow or disallow page breaks.

\lua_now:n { piton.CountLines(token.scan_argument()) } { ##1 }

If the final user has used both left-margin=auto and line-numbers, we have to compute the width of the maximal number of lines at the end of the environment to fix the correct value to left-margin.

\bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool

\bool_if:NTF \l_@@_all_line_numbers_bool

\hbox_set:Nn \l_tmpa_box

\footnotesize
\int_to_arabic:n { \g_@@_visual_line_int + \l_@@_nb_lines_int }
\lua_now:n

\bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
\vtop \bgroup
\lua_now:e { piton.GobbleParse ( \int_use:N \l_@@_gobble_int , token.scan_argument() ) }
\vspace { 2.5 pt }
\egroup
\bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
\@@_width_to_aux:

The following \end{#1} is only for the groups and the stack of environments of LaTeX.

\end { #1 }
\@@_write_aux:

We can now define the new environment.

We are still in the definition of the command \NewPitonEnvironment...

\NewDocumentEnvironment { #1 } { #2 }

\@@_pre_env:
The following code is for technical reasons. We want to change the catcode of `\^^M` before catching the arguments of the new environment we are defining. Indeed, if not, we will have problems if there is a final optional argument in our environment (if that final argument is not used by the user in an instance of the environment, a spurious space is inserted, probably because the `\^^M` is converted to space).

\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }

This is the end of the definition of the command \NewPitonEnvironment.

\NewPitonEnvironment { Piton } { } { } { }

6.2.8 The styles

The following command is fundamental: it will be used by the Lua code.

\NewDocumentCommand { \PitonStyle } { m } { \use:c { pitonStyle #1 } }

The following command takes its argument by curryfication.

\NewDocumentCommand { \SetPitonStyle } { } { \keys_set:nn { piton / Styles } }

\cs_new_protected:Npn \@@_math_scantokens:n #1
{ \normalfont \scantextokens { $#1$ } }

\keys_define:nn { piton / Styles } {%
  String.Interpol .tl_set:c = pitonStyle String.Interpol ,
  String.Interpol .value_required:n = true ,
  FormattingType .tl_set:c = pitonStyle FormattingType ,
  FormattingType .value_required:n = true ,
  Dict.Value .tl_set:c = pitonStyle Dict.Value ,
  Dict.Value .value_required:n = true ,
  Name.Decorator .tl_set:c = pitonStyle Name.Decorator ,
  Name.Decorator .value_required:n = true ,
  Name.Function .tl_set:c = pitonStyle Name.Function ,
  Name.Function .value_required:n = true ,
  Keyword .tl_set:c = pitonStyle Keyword ,
  Keyword .value_required:n = true ,
  Keyword.Constant .tl_set:c = pitonStyle Keyword.Constant ,
  Keyword.constant .value_required:n = true ,
  String.Doc .tl_set:c = pitonStyle String.Doc ,
  String.Doc .value_required:n = true ,
  Interpol.Inside .tl_set:c = pitonStyle Interpol.Inside ,
  Interpol.Inside .value_required:n = true ,
  String.Long .tl_set:c = pitonStyle String.Long ,
  String.Long .value_required:n = true ,
  String.Short .tl_set:c = pitonStyle String.Short ,
  String.Short .value_required:n = true ,
  String .meta:n = { String.Long = #1 , String.Short = #1 } ,
  Comment.Math .tl_set:c = pitonStyle Comment.Math ,
  Comment.Math .default:n = \@@_math_scantokens:n ,
  Comment.Math .initial:n = ,
  Comment .tl_set:c = pitonStyle Comment ,
  .value_required:n = true ,
  InitialValues .tl_set:c = pitonStyle InitialValues ,
}
6.2.9 The initial style

The initial style is inspired by the style “manni” of Pygments.

```latex
\SetPitonStyle
{
  Comment = \color[HTML]{0099FF} \itshape ,
  Exception = \color[HTML]{CC0000} ,
  Keyword = \color[HTML]{006699} \bfseries ,
  InitialValues = \color[HTML]{0099FF} \itshape ,
}
```

```latex
\msg_new:nnn { piton } { Unknown-key-for-SetPitonStyle } { The-style-\l_keys_key_str-is-unknown.\ This-key-will-be-ignored.\ The-available-styles-are-(in-alphabetic-order):- Comment,- Comment.LaTeX,- Dict.Value,- Exception,- InitialValues,- Keyword,- Keyword.Constant,- Name.Builtin,- Name.Class,- Name.Decorator,- Name.Function,- Name.Namespace,- Number,- Operator,- Operator.Word,- String,- String.Doc,- String.Long,- String.Short,-and- String.Interpol. }
```
The last style Post.Function should be considered as an “internal style” (not available for the final user). If the key math-comments has been used at load-time, we change the style Comment.Math which should be considered only at an “internal style”. However, maybe we will document in a future version the possibility to write change the style locally in a document).

\bool_if:NT \c_@@_math_comments_bool { \SetPitonStyle { Comment.Math } }

6.2.10 Security

\AddToHook { env / piton / begin } { \msg_fatal:nn { piton } { No~environment~piton } }
\msg_new:nnn { piton } { No~environment~piton } { There~is~no~environment~piton!\}
There-is-an-environment-{Piton}-and-a-command-
\token_to_str:N \piton\ but-there-is-no-environment-
{piton}.~This~error~is~fatal.

6.3 The Lua part of the implementation

\ExplSyntaxOff
\RequirePackage{luacode}

The Lua code will be loaded via a \{luacode\*} environment. Thei environment is by itself a Lua block and the local declarations will be local to that block. All the global functions (used by the L3 parts of the implementation) will be put in a Lua table piton.

\begin{luacode*}
piton = piton or { }
if piton.comment_latex == nil then piton.comment_latex = "\" end
if piton.comment_latex == "#" then piton.comment_latex = "\" end

6.3.1 Special functions dealing with LPEG

We will use the Lua library lpeg which is built in LuaTeX. That’s why we define first aliases for several functions of that library.

local Cf, Cs = lpeg.Cf, lpeg.Cs
The function \( Q \) takes in as argument a pattern and returns a \texttt{lpeg} which does a capture of the pattern. That capture will be sent to LaTeX with the catcode “other” for all the characters: it’s suitable for elements of the Python listings that \texttt{piton} will typeset verbatim (thanks to the catcode “other”).

\begin{verbatim}
local function Q(pattern)
  return Ct ( Cc ( luahtextbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
end
\end{verbatim}

The function \( L \) takes in as argument a pattern and returns a \texttt{lpeg} which does a capture of the pattern. That capture will be sent to LaTeX with standard LaTeX catcodes for all the characters: the elements captured will be formatted as normal LaTeX codes. It’s suitable for the “comment LaTeX” in the environments \{Piton\} and the elements between “escape-inside”. That function won’t be much used.

\begin{verbatim}
local function L(pattern)
  return Ct ( C ( pattern ) )
end
\end{verbatim}

The function \( Lc \) (the \( c \) is for \textit{constant}) takes in as argument a string and returns a \texttt{lpeg} with does a \textit{constant capture} which returns that string. The elements captured will be formatted as L3 code. It will be used to send to LaTeX all the formatting LaTeX instructions we have to insert in order to do the syntactic highlighting (that’s the main job of \texttt{piton}). That function will be widely used.

\begin{verbatim}
local function Lc(string)
  return Cc ( { luahtextbase.catcodetables.expl , string } )
end
\end{verbatim}

The function \( K \) creates a \texttt{lpeg} which will return as capture the whole LaTeX code corresponding to a Python chunk (that is to say with the LaTeX formatting instructions corresponding to the syntactic nature of that Python chunk). The first argument is a pattern (that is to say a \texttt{lpeg} without capture) and the second element is a Lua string corresponding to the name of a \texttt{piton} style. If the second argument is not present, the function \( K \) behaves as the function \( Q \) does.

\begin{verbatim}
local function K(pattern, style)
  if style
    then
    return
      Lc ( "\{\PitonStyle{\textcolor{red}{\textbf{\texttt{Keyword}}}}{\textcolor{red}{\textbf{\texttt{\colorbox{yellow}{\texttt{text to format}}}}}} \}
      * Q ( pattern )
      * Lc ( "\})" )
    else
    return Q ( pattern )
  end
end
\end{verbatim}

The formatting commands in a given \texttt{piton} style (eg. the style \texttt{Keyword}) may be semi-global declarations (such as \texttt{\bfseries} or \texttt{\slshape}) or LaTeX macros with an argument (such as \texttt{\fbox} or \texttt{\colorbox{yellow}}). In order to deal with both syntaxes, we have used two pairs of braces: \texttt{\{\PitonStyle{\textcolor{red}{\textbf{\texttt{Keyword}}}}{\textcolor{red}{\textbf{\texttt{\colorbox{yellow}{\texttt{text to format}}}}}}\}}.

The following \texttt{lpeg} catches the Python chunks which are in LaTeX escapes (and that chunks will be considered as normal LaTeX constructions). We recall that \texttt{piton.begin.escape} and \texttt{piton.end.escape} are Lua strings corresponding to the key \texttt{escape-inside}\footnote{The \texttt{piton} key \texttt{escape-inside} is available at load-time only.}. Since the elements that will be catched must be sent to LaTeX with standard LaTeX catcodes, we put the capture (done by the function \( C \)) in a table (by using \( Ct \), which is an alias for \texttt{lpeg.Ct}) without number of catcode table at the first component of the table.

\begin{verbatim}
local Escape =
P(piton_begin_escape)
  * L ( ( 1 - P(piton_end_escape) ) ^ 1 )
  * P(piton_end_escape)
\end{verbatim}
The following line is mandatory.

```lpeg
lpeg.locale(lpeg)
```

### 6.3.2 The LPEG Syntax

```python
local alpha, digit, space = lpeg.alpha, lpeg.digit, lpeg.space
```

Remember that, for LPEG, the Unicode characters such as à, â, ç, etc. are in fact strings of length 2 (2 bytes) because lpeg is not Unicode-aware.

```python
local letter = alpha + P "_."
+ P "â" + P "â" + P "ç" + P "é" + P "è" + P "ê" + P "è" + P "é" + P "ê"
+ P "ô" + P "û" + P "ü" + P "Â" + P "À" + P "Ç" + P "É" + P "Ê" + P "È"
+ P "Æ" + P "Ï" + P "Î" + P "Œ" + P "Œ" + P "œ" + P "Œ" + P "Œ"
```

```python
local alphanum = letter + digit
```

The following LPEG `identifier` is a mere pattern (that is to say more or less a regular expression) which matches the Python identifiers (hence the name).

```python
local identifier = letter + alphanum ^ 0
```

On the other hand, the LPEG `Identifier` (with a capital) also return a capture.

```python
local Identifier = K ( identifier )
```

By convention, we will use names with an initial capital for LPEG which return captures.

Here is the first use of our function `K`. That function will be used to construct LPEG which capture Python chunks for which we have a dedicated `piton` style. For example, for the numbers, `piton` provides a style which is called `Number`. The name of the style is provided as a Lua string in the second argument of the function `K`. By convention, we use single quotes for delimiting the Lua strings which are names of `piton` styles (but this is only a convention).

```python
local Number =
K ( ( digit "1 * P "." * digit"0 + digit"0 * P "." * digit"1 + digit"1 )
* ( S "eE" * S "+-" ^ -1 * digit"1 ) ^ -1
+ digit"1 ,
'Number' )
```

We recall that `piton.begin_escape` and `piton_end_escape` are Lua strings corresponding to the key `escape-inside`\(^\text{16}\). Of course, if the final user has not used the key `escape-inside`, these strings are empty.

```python
local Word
if piton_begin_escape ~= ''
then Word = K ( ( ( 1 - space - P(piton_begin_escape) - P(piton_end_escape) )
- S "\"\"\"r[()]" - digit ) ^ 1 )
else Word = K ( ( ( 1 - space ) - S "\"\"\"r[()]" - digit ) ^ 1 )
end

local Space = K ( ( space - P "\"r" ) ^ 1 )
local SkipSpace = K ( ( space - P "\"r" ) ^ 0 )
local Punct = K ( S ",.;;:!")
local Tab = P "\"t" * Lc ( 'l_@@_tab_tl' )
```

\(^{16}\)The `piton` key `escape-inside` is available at load-time only.
local SpaceIndentation = Lc ( '\@@_an_indentation_space:' ) * K " 

The following LPEG EOL is for the end of lines.

local EOL = P \r* ( space^0 * -1 ) +

We recall that each line in the Python code we have to parse will be sent back to LaTeX between a pair \@@_begin_line:–\@@_end_line:.

local Delim = K ( S "[()]" )

Some strings of length 2 are explicit because we want the corresponding ligatures in the font Fira Code to be active.

local Operator = K ( P "!=" + P "<>" + P "==" + P "<" + P ">>" + P "<=" + P ">=" + P ":="
+ P "///" + P "**" + S "--/+*%<<>$.?!

, 'Operator'
)

local OperatorWord = K ( P "in" + P "is" + P "and" + P "or" + P "not" , 'Operator.Word')

local Keyword = K ( P "as" + P "assert" + P "break" + P "case" + P "class" + P "continue"
+ P "def" + P "del" + P "elif" + P "else" + P "except" + P "exec"
+ P "finally" + P "for" + P "from" + P "global" + P "if" + P "import"
+ P "lambda" + P "non local" + P "pass" + P "return" + P "try"
+ P "while" + P "with" + P "yield" + P "yield from" ,

'Keyword'
)

local Exception = K ( "ArithmeticError" + P "AssertionError" + P "AttributeError"
+ P "BaseException" + P "BufferError" + P "BytesWarning" + P "DeprecationWarning"

\footnote{Remember that the \@@_end_line: must be explicit because it will be used as marker in order to delimit the argument of the command \@@_begin_line:}
In Python, a "decorator" is a statement whose begins by `@` which patches the function defined in the following statement.

The following LPEG `DefClass` will be used to detect the definition of a new class (the name of that new class will be formatted with the piton style `Name.Class`).

Example: `class myclass:
    local DefClass =
    K ( P "class" , 'Keyword' ) * Space * K ( identifier , 'Name.Class' )
If the word class is not followed by a identifier, it will be catched as keyword by the LPEG `Keyword` (useful if we want to type a list of keywords).

The following LPEG `ImportAs` is used for the lines beginning by `import`. We have to detect the potential keyword as because both the name of the module and its alias must be formatted with the piton style `Name.Namespace`.

Example: `import numpy as np`
Moreover, after the keyword `import`, it’s possible to have a comma-separated list of modules (if the keyword `as` is not used).

Example: `import math, numpy`

Be careful: there is no commutativity of `+` in the previous expression.

The LPEG `FromImport` is used for the lines beginning by `from`. We need a special treatment because the identifier following the keyword `from` must be formatted with the piton style `Name.Namespace` and the following keyword `import` must be formatted with the piton style `Keyword` and must not be catched by the LPEG `ImportAs`. 
Example:
```
from math import pi
```

The strings of Python

For the strings in Python, there are four categories of delimiters (without counting the prefixes for f-strings and raw strings). We will use, in the names of our LPEG, prefixes to distinguish the LPEG dealing with that categories of strings, as presented in the following tabular.

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>'text'</td>
</tr>
<tr>
<td>Long</td>
<td>&quot;text&quot;</td>
</tr>
</tbody>
</table>

First, we define LPEG for the interpolations in the f-strings. Here is an example of a f-string with an interpolation and a format instruction in that interpolation:
```
f'Total price: {total+1:.2f} €'
```

The following LPEG `SingleShortInterpol` (and the three variants) will catch the whole interpolation, included the braces, that is to say, in the previous example: `{total+1:.2f}

The following LPEG catches a space (U+0032) and replace it by \l_@@_space_tl. It will be used in the short strings. Usually, \l_@@_space_tl will contain a space and therefore there won’t be difference. However, when the key show-spaces is in force, \l_@@_space_tl will contain ␣ (U+2423) in order to visualize the spaces.
```
local VisualSpace = P " \l_@@_space_tl"
```

Now, we define LPEG for the parts of the strings which are not in the interpolations.
local DoubleShortPureString = 
( K ( ( P "\\\"" + P "{{" + P "}}" + 1 - S " {\"" ) ^ 1 ) + VisualSpace ) ^ 1

local SingleLongPureString = 
K ( ( 1 - P "''''" - S "{"\"x" ) ^ 1 )

local DoubleLongPureString = 
K ( ( 1 - P "\\\"\"" - S " {\"" ) ^ 1 )

The interpolations beginning by % (even though there is more modern technics now in Python).

local PercentInterpol = 
K ( P "%"
* ( P "\" + alphanum ^ * P "\" ) ^ -1
* ( S "\"#0 +" ) ^ 0
* ( digit ~ 1 + P "*" ) ^ -1
* ( P "." * ( digit ~ 1 + P "*" ) ) ^ -1
* ( S "HlL" ) ~ -1
* S "sdFeExorgiGauc%",
'String.Interpol'
)

We can now define the LPEG for the four kinds of strings. It’s not possible to use our function K because of the interpolations which must be formatted with another piton style that the rest of the string.

local SingleShortString =
Lc ( "\{\\PitonStyle{String.Short}\{" 
*
First, we deal with the f-strings of Python, which are prefixed by f or F.

K ( P "f" + P "F" )
* ( SingleShortInterpol + SingleShortPureString ) ^ 0
* K ( P "\""
+

Now, we deal with the standard strings of Python, but also the “raw strings”.

K ( P "\"" + P "r" + P "R" )
* ( K ( ( P "\\\"" + 1 - S " \"%" ) ^ 1 )
  + VisualSpace
  + PercentInterpol
  + K ( P "\""
  ) ~ 0
  * K ( P "\""
  )
  * Lc ( "\}"
)

local DoubleShortString =
Lc ( "\{\PitonStyle{String.Short}\{" 
*
K ( P "f\"" + P "F\""
* ( DoubleShortInterpol + DoubleShortPureString ) ~ 0
* K ( P "\""
+

19 The interpolations are formatted with the piton style Interpol.Inside. The initial value of that style is \@_piton:n which means that the interpolations are parsed once again by piton.
local ShortString = SingleShortString + DoubleShortString

Of course, it’s more complicated for “longs strings” because, by definition, in Python, those strings may be broken by an end on line (which is catched by the LPEG EOL).

local SingleLongString =
Lc "\{"\PitonStyle{String.Long}\{""*
* ( K ( S "fF" * P "''''" )
* ( SingleLongInterpol + SingleLongPureString ) ^ 0
* Lc "}")*
* ( EOL
 +
 Lc "\{"\PitonStyle{String.Long}\{""
 * ( SingleLongInterpol + SingleLongPureString ) ^ 0
 * Lc "}")"
 * EOL
 ) ^ 0
 * Lc "\{"\PitonStyle{String.Long}\{""
 * ( SingleLongInterpol + SingleLongPureString ) ^ 0
 +
 K ( ( S "rR" ) ^ -1 * P "''''" 
 * ( 1 - P "''''" - P "\r" ) ^ 0 )
 * Lc "}")*
 ( Lc "\{"\PitonStyle{String.Long}\{""
 * K ( ( 1 - P "''''" - P "\r" ) ^ 0 )
 * Lc "}")
 ) ^ 0
 * Lc "\{"\PitonStyle{String.Long}\{""
 * K ( ( 1 - P "''''" - P "\r" ) ^ 0 )
 )
 * Lc "}")"

local DoubleLongString =
Lc "\{"\PitonStyle{String.Long}\{""
* ( K ( S "fF" * P "\\\\\\\\\\"
* ( DoubleLongInterpol + DoubleLongPureString ) ^ 0
* Lc "}")*
* ( EOL
 +
 Lc "\{"\PitonStyle{String.Long}\{""
 * ( DoubleLongInterpol + DoubleLongPureString ) ^ 0
 * Lc "}")"
 * EOL
 ) ^ 0
 * Lc "\{"\PitonStyle{String.Long}\{""
 * ( DoubleLongInterpol + DoubleLongPureString ) ^ 0
 +
 K ( ( S "rR" ) ^ -1 * P "\\\\\\\\\\"
 * ( 1 - P "\\\\\\\\\\" - P "\r" ) ^ 0 )
 * Lc "}")*
 ( Lc "\{"\PitonStyle{String.Long}\{""
local LongString = SingleLongString + DoubleLongString

We have a LPEG for the Python docstrings. That LPEG will be used in the LPEG DeffFunction which deals with the whole preamble of a function definition (which begins with `def`).

```lpeg
local StringDoc =
  K ( P "\"\"" , 'String.Doc' )
* ( K ( (1 - P "\"\"" - P \r" ) ^ 0 , 'String.Doc' ) * EOL * Tab ^0 ) ^ 0
* K ( ( 1 - P "\"\"" ) ^ 0 * P "\"\"" , 'String.Doc' )
```

The comments in the Python listings  We define different LPEG dealing with comments in the Python listings.

```lpeg
local CommentMath =
P "$" * K ( ( 1 - S "$\r" ) ^ 1 , 'Comment.Math' ) * P "$"
local Comment =
Lc ( "\{\PitonStyle{Comment}}{{\ignorespaces
* K ( P "#" )
* ( CommentMath + K ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0
* Lc ( "\}" )
* ( EOL + -1 )
```

The following LPEG CommentLaTeX is for what is called in that document the “LaTeX comments”. Since the elements that will be catched must be sent to LaTeX with standard LaTeX catcodes, we put the capture (done by the function `C`) in a table (by using `Ct`, which is an alias for `lpeg.Ct`).

```lpeg
local CommentLaTeX =
P(piton.comment_latex)
* Lc "\{\PitonStyle{Comment.LaTeX}{\ignorespaces"
* L ( ( 1 - P \r" ) ^ 0 )
* Lc "\}"
* ( EOL + -1 )
```

DefFunction  The following LPEG Expression will be used for the parameters in the `argspec` of a Python function. It’s necessary to use a grammar because that pattern mainly checks the correct nesting of the delimiters (and it’s known in the theory of formal languages that this can’t be done with regular expressions stricto sensu only).

```lpeg
local Expression =
P { "E" ,
  E = ( 1 - S "\{}\[]\r," ) ^ 0
  * ( ( P "\{}\[]\rE" * P "\r" )
    + P "\"\"" * V "F" * P "\r"
    + P "[" * V "F" * P "]" ) * ( 1 - S "\{}\[]\r," ) ^ 0
  ) ^ 0
  ,
  F = ( 1 - S "\{}\[]\r\"" ) ^ 0
  * ( ( P "\"" * (P \"\"" + 1 - S\"\"\r" ) ^0 * P "\""
    + P \"\"" * (P \"\"" + 1 - S\"\"\r" ) ^0 * P \"\"
    + P "\"" * V "F" * P "\r" )
```
We will now define a **LPEG** `Params` that will catch the list of parameters (that is to say the *argspec*) in the definition of a Python function. For example, in the line of code

```python
def MyFunction(a,b,x=10,n:int): return n
```

the **LPEG** `Params` will be used to catch the chunk `a,b,x=10,n:int`. Or course, a `Params` is simply a comma-separated list of `Param`, and that's why we define first the **LPEG** `Param`.

```plaintext
local Param =
  SkipSpace * Identifier * SkipSpace
  *
  (  K ( P "=" * Expression , 'InitialValues' )
    + K ( P ":" ) * SkipSpace * K ( letter^1 , 'Name.Type' )
  ) ^ -1
```

```plaintext
local Params = ( Param * ( K "," * Param ) ^ 0 ) ^ -1
```

The following **LPEG** `DefFunction` catches a keyword `def` and the following name of function *but also everything else until a potential docstring*. That's why this definition of **LPEG** must occur (in the file `piton.sty`) after the definition of several other **LPEG** such as `Comment`, `CommentLaTeX`, `Params`, `StringDoc`...

```plaintext
local DefFunction =
  K ( P "def" , 'Keyword' )
  * Space
  * K ( identifier , 'Name.Function' )
  * SkipSpace
  * K ( P "(" ) * Params * K ( P ")" )
  * SkipSpace
  * ( K ( P "->" ) * SkipSpace * K ( identifier , 'Name.Type' ) ) ^ -1
```

Here, we need a `piton` style `Post.Function` which will be linked to `\@@_piton:n` (that means that the capture will be parsed once again by `piton`). We could avoid that kind of trick by using a non-terminal of a grammar but we have probably here a better legibility.

```plaintext
* K ( ( 1 - S ":\r" )^0 , 'Post.Function' )
* K ( P ":" )
* ( SkipSpace
  * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
  * Tab ^ 0
  * SkipSpace
  * StringDoc ^ 0 -- there may be additionnal docstrings
) ^ -1
```

Remark that, in the previous code, `CommentLaTeX` *must* appear before `Comment`: there is no commutativity of the addition for the parsing expression grammars (PEG).

If the word `def` is not followed by a identifier and parenthesis, it will be catched as keyword by the **LPEG** `Keyword` (useful if, for example, the final user wants to speak of the keyword `def`).
The dictionaries of Python. We have LPEG dealings with dictionaries of Python because, in typesettings of explicit Python dictionaries, one may prefer to have all the values formatted in black (in order to see more clearly the keys which are usually Python strings). That’s why we have a piton style Dict.Value.

The initial value of that piton style is \@@_piton:n, which means that the value of the entry of the dictionary is parsed once again by piton (and nothing special is done for the dictionary). In the following example, we have set the piton style Dict.Value to \color{black}:

```
mydict = { 'name' : 'Paul', 'sex' : 'male', 'age' : 31 }
```

At this time, this mechanism works only for explicit dictionaries on a single line!

```lpeg
local ItemDict =
  ShortString * SkipSpace * K ( P ":" ) * K ( Expression, 'Dict.Value' )
local ItemOfSet = SkipSpace * ( ItemDict + ShortString ) * SkipSpace
local Set =
  K ( P "{" )
  * ItemOfSet * ( K ( P "," ) * ItemOfSet ) ^ 0
  * K ( P "}" )
```

The main LPEG SyntaxPython is the main LPEG of the package piton. We have written an auxiliary LPEG SyntaxPythonAux only for legibility.

```lpeg
local SyntaxPythonAux =
We recall that each line in the Python code to parse will be sent back to LaTeX between a pair \@@_begin_line: – \@@_end_line: 20.

```
\@@_begin_line: \" \$
```
```
\@@_end_line: \" \$
```

Remember that the \@@_end_line: must be explicit because it will be used as marker in order to delimit the argument of the command \@@_begin_line:

---

20Remember that the \@@_end_line: must be explicit because it will be used as marker in order to delimit the argument of the command \@@_begin_line:
We have written an auxiliary LPEG SyntaxPythonAux for legibility only.

```
local SyntaxPython = Ct ( SyntaxPythonAux )
```

### 6.3.3 The function Parse

The function `Parse` is the main function of the package `piton`. It parses its argument and sends back to LaTeX the code with interlaced formatting LaTeX instructions. In fact, everything is done by the LPEG `SyntaxPython` which returns as capture a Lua table containing data to send to LaTeX.

```
function piton.Parse(code)
  local t = SyntaxPython : match ( code ) -- match is a method of the LPEG
  for _, s in ipairs(t) do tex.tprint(s) end
end
```

The following command will be used by the user commands \`piton`. For that command, we have to undo the duplication of the symbols `#`.

```
function piton.pitonParse(code)
  local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
  return piton.Parse(s)
end
```

The function `ParseFile` will be used by the LaTeX command \`PitonInputFile`. That function merely reads the whole file (that is to say all its lines) and then apply the function `Parse` to the resulting Lua string.

```
function piton.ParseFile(name,first_line,last_line)
  s = ''
  local i = 0
  for line in io.lines(name) do i = i + 1
    if i >= first_line then s = s .. '\r' .. line
    end
    if i >= last_line then break end
  end
  piton.Parse(s)
end
```

### 6.3.4 The preprocessors of the function Parse

We deal now with preprocessors of the function `Parse` which are needed when the “gobble mechanism” is used.

The function `gobble` gobbles `n` characters on the left of the code. It uses a LPEG that we have to compute dynamically because if depends on the value of `n`.

```
local function gobble(n,code)
  if n==0 then return code else
    return Cf (
      Cc ( "" ) *
      ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
      * ( C ( P "\r" ) *
          ( 1 - P "\r" ) ^ (-n)
      * C ( ( 1 - P "\r" ) ^ 0 ) ) ^ 0 ,
```
concat
) : match ( code )
end
end

The following function add will be used in the following LPEG AutoGobbleLPEG and EnvGobbleLPEG.

local function add(acc,new_value)
  return acc + new_value
end

The following LPEG returns as capture the minimal number of spaces at the beginning of the lines of code. The main work is done by two fold captures (lpeg.Cf), one using add and the other (encompassing the previous one) using math.min as folding operator.

local AutoGobbleLPEG =
  ( space ^ 0 * P "\r" ) ^ -1
  * Cf ( ( P " " ) ^ 0 * P "\r"
        +
        Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
        * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * P "\r"
        ) ^ 0
We don’t take into account the empty lines (with only spaces).

(P = " ") ^ 0 * P "\r"
  +
  Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
  * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
  math.min

Now for the last line of the Python code...

* ( Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
  * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1

The following LPEG returns as capture the number of spaces at the last line, that is to say before the \end{Piton} (and usually it’s also the number of spaces before the corresponding \begin{Piton} because that’s the traditional way to indent in LaTeX). The main work is done by a fold capture (lpeg.Cf) using the function add as folding operator.

local EnvGobbleLPEG =
  ( ( 1 - P "\r" ) ^ 0 * P "\r" ) ^ 0
  * Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add ) * -1

function piton.GobbleParse(n,code)
  if n==-1
    then n = AutoGobbleLPEG : match(code)
  else if n==-2
    then n = EnvGobbleLPEG : match(code)
  end
  end
  piton.Parse(gobble(n,code))
end

6.3.5 To count the number of lines

function piton.CountLines(code)
  local count = 0
  for i in code : gmatch ( "\r" ) do count = count + 1 end
  tex.sprint(\luatexbase.catcodetables.expl ,
    '\\int_set:Nn \l_@@_nb_lines_int {' .. count .. '}')
end
function piton.CountNonEmptyLines(code)
  local count = 0
  count =
    ( Cf ( Cc(0) *
      ( P " " ) ^ 0 * P "\r"
      + ( 1 - P "\r" ) ^ 0 * P "\r" * Cc(1)
    ) ^ 0
    * (1 - P "\r" ) ^ 0 ,
    add
    ) * -1 ) : match (code)
  tex.sprint(
    luatexbase.catcodetables.expl ,
    '\int_set:Nn \l_@@_nb_non_empty_lines_int {' .. count .. '}' )
end

function piton.CountLinesFile(name)
  local count = 0
  for line in io.lines(name) do count = count + 1 end
  tex.sprint(
    luatexbase.catcodetables.expl ,
    '\int_set:Nn \l_@@_nb_lines_int {' .. count .. '}' )
end

function piton.CountNonEmptyLinesFile(name)
  local count = 0
  for line in io.lines(name)
    do if not ( ( ( P " " ) ^ 0 * -1 ) : match ( line ) )
      then count = count + 1
    end
  end
  tex.sprint(
    luatexbase.catcodetables.expl ,
    '\int_set:Nn \l_@@_nb_non_empty_lines_int {' .. count .. '}' )
end
\end{luacode*}

7 History

Changes between versions 0.6 and 0.7
New keys resume, splittable and background-color in \PitonOptions. The file piton.lua has been embedded in the file piton.sty. That means that the extension piton is now entirely contained in the file piton.sty.

Changes between versions 0.7 and 0.8
New keys footnote and footnotehyper at load-time. New key left-margin.

Changes between versions 0.8 and 0.9
New key tab-size. Integer value for the key splittable.
Changes between versions 0.9 and 0.95

New key show-spaces.
The key left-margin now accepts the special value auto.
New key latex-comment at load-time and replacement of ## by #>
New key math-comments at load-time.
New keys first-line and last-line for the command \InputPitonFile.

Changes between versions 0.95 and 0.99

New key break-lines to allow breaks of the lines of code (and other keys to customize the appearance).