The package piton

F. Pantigny
fpantigny@wanadoo.fr
August 26, 2023

Abstract

The package piton provides tools to typeset computer listings in Python, OCaml and C with syntactic highlighting by using the Lua library LPEG. It requires LuaLaTeX.

1 Presentation

The package piton uses the Lua library LPEG\(^1\) for parsing Python OCaml or C listings and typesets them with syntactic highlighting. Since it uses Lua code, it works with lualatex only (and won’t work with the other engines: latex, pdflatex and 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 piton, with the environment \texttt{\{Piton\}}.

\begin{verbatim}
from math import pi

def arctan(x,n=10):
    """Compute the mathematical value of arctan(x)
    \textit{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)\(^2\)
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
\end{verbatim}

2 Installation

The package piton is contained in two files: piton.sty and piton.lua (the LaTeX file piton.sty loaded by \texttt{\{usepackage\} will load the Lua file piton.lua}). Both files must be in a repertory where LaTeX will be able to find them, for instance in a texmf tree. However, the best is to install piton with a TeX distribution such as MiKTeX, TeX Live or MacTeX.

\footnote{This document corresponds to the version 2.1 of piton, at the date of 2023/08/26.}

\footnote{LPEG is a pattern-matching library for Lua, written in C,\footnote{http://www.inf.puc-rio.br/~roberto/lpeg/} based on \textit{parsing expression grammars}: \texttt{http://www.inf.puc-rio.br/~roberto/lpeg/}}

\footnote{This LaTeX escape has been done by beginning the comment by \#>.
3 Use of the package

3.1 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.

3.2 Choice of the computer language

In current version, the package piton supports three computer languages: Python, OCaml and C (in fact C++).

By default, the language used is Python.

It’s possible to change the current language with the command \PitonOptions and its key language: \PitonOptions{language = C}.

In what follows, we will speak of Python, but the features described also apply to the other languages.

3.3 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:

\begin{verbatim}
\piton{def square(x): return x*x} 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. 4.3 p. 7.

- The command \PitonInputFile is used to insert and typeset a external file. It’s possible to insert only a part of the file: cf. part 5.2, p. 9.

3.4 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,
    but the command \ \ is provided to force the insertion of a space;
  - it’s not possible to use % inside the argument,
    but the command \% is provided to insert a \;
  - the braces must be appear by pairs correctly nested
    but the commands \( and \) are also provided for individual braces;
the LaTeX commands are fully expanded and not executed, so it’s possible to use \ to insert a backslash.

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

Examples:

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

It’s possible to use the command \piton in the arguments of a LaTeX command.

- **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.

Examples:

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

4 Customization

With regard to the font used by piton in its listings, it’s only the current monospaced font. The package piton merely uses internally the standard LaTeX command \texttt.

4.1 The keys of 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. These keys may also be applied to an individual environment {Piton} (between square brackets).

- The key **language** specifies which computer language is considered (that key is case-insensitive). Three values are allowed: Python, OCaml and C. The initial value is Python.
- The key **gobble** takes in as value a positive integer \(n\): the first \(n\) characters are discarded (before the process of highlightning 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.

---

3That concerns the commands beginning with a backslash but also the active characters (with catcode equal to 13).
4For example, it’s possible to use the command \piton in a footnote. Example: \(s = 'A string'\).
5We remind that a LaTeX environment is, in particular, a TeX group.
• The key `line-numbers` activates the line numbering in the environments `{Piton}` and in the listings resulting from the use of `\PitonInputFile`.

**New 2.1** In fact, the key `line-numbers` has several subkeys.

- With the key `line-numbers/skip-empty-lines`, the empty lines are considered as non-existent for the line numbering (if the key `/absolute` is in force, the key `/skip-empty-lines` is no-op in `\PitonInputFile`). The initial value of that key is true (and not false).\(^6\)
- With the key `line-numbers/label-empty-lines`, the labels (that is to say the numbers) of the empty lines are displayed. If the key `/skip-empty-line` is in force, the clé `/label-empty-lines` is no-op. The initial value of that key is true.
- With the key `line-numbers/absolute`, in the listings generated in `\PitonInputFile`, the numbers of the lines displayed are absolute (that is to say: they are the numbers of the lines in the file). That key may be useful when `\PitonInputFile` is used to insert only a part of the file (cf. part 5.2, p. 9). The key `/absolute` is no-op in the environments `{Piton}`.
- The key `line-numbers/start` requires that the line numbering begins to the value of the key. That key is not available in `\PitonOptions`.
- With the key `line-numbers/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 `line-numbers/sep` is the horizontal distance between the numbers of lines (inserted by `line-numbers`) and the beginning of the lines of code. The initial value is 0.7 em.

For convenience, a mechanism of factorisation of the prefix `line-numbers` is provided. That means that it is possible, for instance, to write:

\[
\PitonOptions
\{
  line-numbers =
  \{
    skip-empty-lines = false ,
    label-empty-lines = false ,
    sep = 1 \text{ em}
  
  \}
\}
\]

• The key `left-margin` corresponds to a margin on the left. That key may be useful in conjunction with the key `line-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` is in force, a margin will be automatically inserted to fit the numbers of lines. See an example part 6.1 on page 17.

• The key `background-color` sets the background color of the environments `{Piton}` and the listings produced by `\PitonInputFile` (it’s possible to fix the width of that background with the key `width` described below).

The key `background-color` supports also as value a list of colors. In this case, the successive rows are colored by using the colors of the list in a cyclic way.

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

The key `background-color` accepts a color defined «on the fly». For example, it’s possible to write `background-color = [cmyk]{0.1,0.05,0,0}`.

\(^6\)For the language Python, the empty lines in the docstrings are taken into account (by design).
• With the key \texttt{prompt-background-color}, \texttt{piton} adds a color background to the lines beginning with the prompt “>>>” (and its continuation “...”) characteristic of the Python consoles with \texttt{REPL} (\texttt{read-eval-print loop}).

• The key \texttt{width} will fix the width of the listing. That width applies to the colored backgrounds specified by \texttt{background-color} and \texttt{prompt-background-color} but also for the automatic breaking of the lines (when required by \texttt{break-lines}: cf. 5.1.2, p. 8).

That key may take in as value a numeric value but also the special value \texttt{min}. With that value, the width will be computed from the maximal width of the lines of code. Caution: the special value \texttt{min} requires two compilations with LuaLaTeX\footnote{The maximal width is computed during the first compilation, written on the \texttt{aux} file and re-used during the second compilation. Several tools such as \texttt{latexmk} (used by Overleaf) do automatically a sufficient number of compilations.}

For an example of use of \texttt{width=min}, see the section 6.2, p. 17.

• When the key \texttt{show-spaces-in-strings} is activated, the spaces in the short strings (that is to say those delimited by ‘ or ”) are replaced by the character \texttt{␣} (U+2423: open box). Of course, that character U+2423 must be present in the monospaced font which is used.\footnote{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 the package \texttt{fontspec}.}

Example : \texttt{my_string = ‘Very good answer’}

With the key \texttt{show-spaces}, all the spaces are replaced by U+2423 (and no line break can occur on those “visible spaces”, even when the key \texttt{break-lines}\footnote{cf. 5.1.2 p. 8} is in force).

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

### 4.2 The styles

The package `piton` provides the command `\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.\(^{10}\)

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

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. That code uses the command `\highLight` of `lua-ul` (that package requires also the package `luacolor`).

\begin{verbatim}
\SetPitonStyle{ Name.Function = \bfseries \highLight[red!50] }
\end{verbatim}

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

With that setting, we will have: \begin{verbatim} def cube(x) : return x * x * x \end{verbatim}

The different styles are described in the table 7. The initial settings done by `piton` in `piton.sty` are inspired by the style `manni` de Pygments.\(^{11}\)

The command `\PitonStyle` takes in as argument the name of a style and allows to retrieve the value (as a list of LaTeX instructions) of that style.

For example, it’s possible to write `{\PitonStyle{Keyword}{function}}` and we will have the word `function` formatted as a keyword.

The syntax `{\PitonStyle{style}{...}}` is mandatory in order to be able to deal both with the semi-global commands and the commands with arguments which may be present in the definition of the style `style`.

The extension `piton` provides a special style called `UserFunction`. That style applies to the names of the functions previously defined by the user via an instruction Python `def` in one of the previous listings. The initial value of that style is empty, and, therefore, the names of the functions are formatted as standard text (in black). However, it’s possible to change the value of that style, as any other style, with the command `\SetPitonStyle`.

In the following example, we fix as value for that style `UserFunction` the initial value of the the style `Name.Function` (which applies to the name of the functions, at the moment of their definition).

\begin{verbatim}
\SetPitonStyle{UserFunction = \color[HTML]{CC00FF}}
\end{verbatim}

---

\(^{10}\)We remind that a LaTeX environment is, in particular, a TeX group.

\(^{11}\)See: [https://pygments.org/styles/](https://pygments.org/styles/). Remark that, by default, Pygments provides for its style `manni` a colored background whose color is the HTML color #F0F3F3. It’s possible to have the same color in `{Pion}` with the instruction `\PitonOptions{background-color = [HTML]{F0F3F3}}`. 

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

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

As one see, the name `transpose` has been highlighted because it's the name of a Python function previously defined by the user (hence the name `UserFunction` for that style).

Of course, the list of the names of Python functions previously defined is kept in the memory of LuaLaTeX (in a global way, that is to say independently of the TeX groups). The extension `piton` provides a command to clear that list: it's the command `\PitonClearUserFunctions`.

### 4.3 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}`:
\begin{Piton}{}
{\begin{tcolorbox}}
{\end{tcolorbox}}
\end{Piton}{}

If one wishes to format Python code in a box of `tcolorbox`, it's possible to define an environment `{Python}` with the following code (of course, the package `tcolorbox` must be loaded).
\begin{Piton}{}
{\begin{tcolorbox}}
{\end{tcolorbox}}
\end{Piton}{}

With this new environment `{Python}`, it's possible to write:
\begin{Python}
def square(x):
    """Compute the square of a number"
    return x*x
\end{Python}

5 Advanced features

#### 5.1 Page breaks and line breaks

##### 5.1.1 Page breaks

By default, the listings produced by the environment `{Piton}` and the command `\PitonInputFile` are not breakable. However, the command `\PitonOptions` provides the key `splittable` to allow such breaks.
• If the key `splittable` is used without any value, the listings are breakable everywhere.

• If the key `splittable` is used with a numeric value $n$ (which must be a non-negative integer number), the listings are breakable but no break will occur within the first $n$ lines and within the last $n$ lines. Therefore, `splittable=1` is equivalent to `splittable`.

Even with a background color (set by the key `background-color`), the pages breaks are allowed, as soon as the key `splittable` is in force.\textsuperscript{12}

5.1.2 Line breaks

By default, the elements produced by `piton` can’t be broken by an end on line. However, there are keys to allow such breaks (the possible breaking points are the spaces, even the spaces in the Python strings).

• With the key `break-lines-in-piton`, the line breaks are allowed in the command \`piton{...}` (but not in the command \`piton|...|`, that is to say the command \`piton` in verbatim mode).

• With the key `break-lines-in-Piton`, the line breaks are allowed in the environment \{Piton\} (hence the capital letter P in the name) and in the listings produced by \`PitonInputFile`.

• The key `break-lines` is a conjunction of the two previous keys.

The package `piton` provides also several keys to control the appearance on the line breaks allowed by `break-lines-in-Piton`.

• With the key `indent-broken-lines`, the indentation of a broken line is respected at carriage return.

• The key `end-of-broken-line` corresponds to the symbol placed at the end of a broken line. The initial value is: \`\textbackslash`.

• The key `continuation-symbol` corresponds to the symbol placed at each carriage return. The initial value is: +\` (the command \`inserts a small horizontal space).

• The key `continuation-symbol-on-indentation` corresponds to the symbol placed at each carriage return, on the position of the indentation (only when the key `indent-broken-line` is in force). The initial value is: $\hookrightarrow\$.

The following code has been composed with the following tuning:

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

```latex
\begin{Verbatim}
def dict_of_list(l):
    """Converts a list of subrs and descriptions of glyphs in \"\bigleftarrow a dictionary""
    our_dict = {}
    for list_letter in l:
        if (list_letter[0][0:3] == 'dup'):    # if it's a subr
            name = list_letter[0][4:-3]
            print("We treat the subr of number " + name)
        else:
            name = list_letter[0][1:-3]    # if it's a glyph
            print("We treat the glyph of number " + name)
    our_dict[name] = [treat_Postscript_line(k) for k in \bigleftarrow list_letter[1:-1]]
    return dict
\end{Verbatim}
```

\textsuperscript{12}With the key `splittable`, the environments (Piton) are breakable, even within a (breakable) environment of `tcolorbox`. Remind that an environment of `tcolorbox` included in another environment of `tcolorbox` is not breakable, even when both environments use the key `breakable` of `tcolorbox`. 
5.2 Insertion of a part of a file

The command \PitonInputFile inserts (with formatting) the content of a file. In fact, it’s possible to insert only a part of that file. Two mechanisms are provided in this aim.

- It’s possible to specify the part that we want to insert by the numbers of the lines (in the original file).
- **New 2.1** It’s also possible to specify the part to insert with textual markers.

In both cases, if we want to number the lines with the numbers of the lines in the file, we have to use the key line-numbers/absolute.

5.2.1 With line numbers

The command \PitonInputFile supports the keys first-line and last-line in order to insert only the part of file between the corresponding lines. Not to be confused with the key line-numbers/start which fixes the first line number for the line numbering. In a sens, line-numbers/start deals with the output whereas first-line and last-line deal with the input.

5.2.2 With textual markers

**New 2.1**

In order to use that feature, we first have to specify the format of the markers (for the beginning and the end of the part to include) with the keys marker/beginning and marker/end (usually with the command \PitonOptions).

Let us take a practical example.

We assume that the file to include contains solutions to exercises of programmation on the following model.

```python
# [Exercise 1] Iterative version
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
#
```

The markers of the beginning and the end are the strings # [Exercise 1] and #<Exercise 1>. The string “Exercise 1” will be called the label of the exercise (or of the part of the file to be included). In order to specify such markers in piton, we will use the keys marker/beginning and marker/end with the following instruction (the character # of the comments of Python must be inserted with the protected form \#).

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

As one can see, marker/beginning is an expression corresponding to the mathematical function which transforms the label (here Exercise 1) into the the beginning marker (in the example # [Exercise 1]). The string #1 corresponds to the occurrences of the argument of that function, which the classical syntax in TeX. Idem for marker/end.

Now, you only have to use the key range of \PitonInputFile to insert a marked content of the file.

\PitonInputFile[range = Exercise 1]{file_name}
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

The key \texttt{marker/include-line} requires the insertion of the lines containing the markers.

\PitonInputFile[\texttt{marker/include-lines},range = Exercise 1]{file_name}

\texttt{#[Exercise 1]} Iterative version

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

\texttt{#<Exercise 1>}

In fact, there exist also the keys \texttt{begin-range} and \texttt{end-range} to insert several marked contents at the same time.

For example, in order to insert the solutions of the exercises 3 to 5, we will write (if the file has the correct structure!):

\PitonInputFile[\texttt{begin-range = Exercise 3}, \texttt{end-range = Exercise 5}]{file_name}

\subsection{5.3 Highlighting some identifiers}

It's possible to require a changement of formatting for some identifiers with the key \texttt{identifiers} of \PitonOptions.

That key takes in as argument a value of the following format:

\{ \texttt{names = names}, \texttt{style = instructions} \}

- \texttt{names} is a (comma-separated) list of identifier names;

- \texttt{instructions} is a list of LaTeX instructions of the same type as \texttt{piton} “styles” previously presented (cf 4.2 p. 6).

\textit{Caution}: Only the identifiers may be concerned by that key. The keywords and the built-in functions won’t be affected, even if their name is in the list \texttt{names}.

\PitonOptions
\{
    \texttt{identifiers} =
    \{
        \texttt{names} = \{ 11, 12 \},
        \texttt{style} = \texttt{color\{red\}}
    }
\}
\begin{Piton}
def tri(l):
    """Segmentation sort""
    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):
    """Segmentation sort""
    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)

By using the key identifier, it’s possible to add other built-in functions (or other new keywords, etc.) that will be detected by piton.

\PitonOptions
{
    identifiers =
    {
        names = { cos, sin, tan, floor, ceil, trunc, pow, exp, ln, factorial },
        style = \PitonStyle{Name.Builtin}
    }
}

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

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

5.4 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.

One should also remark that, when the extension piton is used with the class beamer, piton detects in \{Piton\} many commands and environments of Beamer: cf. 5.5 p. 14.
5.4.1 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 only in the preamble of the document, allows to choice the characters which, preceded by `#`, will be the syntatic marker.
  
  For example, if the preamble contains the following instruction:
  
  ```latex
  \PitonOptions{comment-latex = LaTeX}
  ```
  
  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 piton style `Comment.LaTeX`.
  
  For example, with `\SetPitonStyle{Comment.LaTeX = \normalfont\color{blue}}`, 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:
  
  ```latex
  \SetPitonStyle{Comment.LaTeX = \color{gray}\#\normalfont\space}
  ```
  
  For other examples of customization of the LaTeX comments, see the part 6.2 p. 17

If the user has required line numbers (with the key `line-numbers`), it’s possible to refer to a number of line with the command `\label` used in a LaTeX comment.13

5.4.2 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`, which is available only in the preamble of the document.

Here is an example, where we have assumed that the preamble of the document contains the instruction `\PitonOptions{math-comment}`:

```
\begin{Piton}
def square(x):
    \begin{Verbatim}[mathematics]
    \begin{Verbatim}[mathematics]
    return x*x # compute $x^2$
\end{Verbatim}
\end{Verbatim}
def square(x):
    return x*x # compute $x^2$
\end{Piton}
```

13That feature is implemented by using a redefinition of the standard command `\label` in the environments `{Piton}`. Therefore, incompatibilities may occur with extensions which redefine (globally) that command `\label` (for example: `varioref`, `refcheck`, `showlabels`, etc.)
5.4.3 The mechanism “escape”

It’s also possible to overwrite the Python listings to insert LaTeX code almost everywhere (but between lexical units, of course). By default, piton does not fix any delimiters for that kind of escape. In order to use this mechanism, it’s necessary to specify the delimiters which will delimit the escape (one for the beginning and one for the end) by using the keys `begin-escape` and `end-escape`, available only in the preamble of the document.

In the following example, we assume that the preamble of the document contains the following instruction:

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

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. That example uses the command `\highLight` of lua-ul (that package requires itself the package luacolor).

\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)

In fact, in that case, it’s probably easier to use the command `@highLight` of lua-ul: that command sets a yellow background until the end of the current TeX group. Since the name of that command contains the character @, it’s necessary to define a synonym without @ in order to be able to use it directly in {Piton}.

\makeatletter
\let\Yellow\@highLight
\makeatother

\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        \Yellow!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 `begin-escape` and `end-escape` 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 \#; such comments are merely called “LaTeX comments” in this document).
5.4.4 The mechanism “escape-math”

The mechanism “escape-math” is very similar to the mechanism “escape” since the only difference is that the elements sent to LaTeX are composed in the math mode of LaTeX.

This mechanism is activated with the keys `begin-escape-math` and `end-escape-math` (which are available only in the preamble of the document).

Despite the technical similarity, the use of the the mechanism “escape-math” is in fact rather different from that of the mechanism “escape”. Indeed, since the elements are composed in a mathematical mode of LaTeX, they are, in particular, composed within a TeX group and therefore, they can’t be used to change the formatting of other lexical units.

In the languages where the character $ does not play a important role, it’s possible to activate that mechanism “escape-math” with the character $:

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

Remark that the character $ must not be protected by a backslash.

However, it’s probably more prudent to use `( et )`.

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

Here is an example of utilisation.

```
\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 += \(\frac{(-1)^k}{2k+1} x^{2k+1}\)
        return s
\end{Piton}
```

5.5 Behaviour in the class Beamer

First remark

Since the environment `{Piton}` catches its body with a verbatim mode, it’s necessary to use the environments `{Piton}` within environments `{frame}` of Beamer protected by the key `fragile`, i.e. beginning with `\begin{frame}[fragile].`

When the package `piton` is used within the class `beamer`, the behaviour of `piton` is slightly modified, as described now.

---

14 Remind that for an environment `{frame}` of Beamer using the key `fragile`, the instruction `\end{frame}` must be alone on a single line (except for any leading whitespace).

15 The extension `piton` detects the class `beamer` and the package `beamerarticle` if it is loaded previously but, if needed, it's also possible to activate that mechanism with the key `beamer` provided by `piton` at load-time: `\usepackage[beamer]{piton}`
5.5.1 \{Piton\} et \PitonInputFile\ are “overlay-aware”

When \texttt{piton} is used in the class \texttt{beamer}, the environment \{Piton\} and the command \PitonInputFile accept the optional argument <...> of Beamer for the overlays which are involved. For example, it’s possible to write:

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

and

\PitonInputFile<2-5>{my_file.py}

5.5.2 Commands of Beamer allowed in \{Piton\} and \PitonInputFile

When \texttt{piton} is used in the class \texttt{beamer}, the following commands of \texttt{beamer} (classified upon their number of arguments) are automatically detected in the environments \{Piton\} (and in the listings processed by \PitonInputFile):

- no mandatory argument: \texttt{\pause}\footnote{One should remark that it’s also possible to use the command \texttt{\pause} in a “LaTeX comment”, that is to say by writing \#: \texttt{\pause}. By this way, if the Python code is copied, it’s still executable by Python.};
- one mandatory argument: \texttt{\action, \alert, \invisible, \only, \uncover and \visible};
- two mandatory arguments: \texttt{\alt};
- three mandatory arguments: \texttt{\temporal}.

In the mandatory arguments of these commands, the braces must be balanced. However, the braces included in short strings\footnote{The short strings of Python are the strings delimited by characters ‘ or the characters “ and not ‘‘ nor “”. In Python, the short strings can’t extend on several lines.} of Python are not considered.

Regarding the functions \texttt{\alt} and \texttt{\temporal} there should be no carriage returns in the mandatory arguments of these functions.

Here is a complete example of file:

\begin{verbatim}
\documentclass{beamer}
\usepackage{piton}
\begin{document}
\begin{frame}[fragile]
\begin{Piton}
def string_of_list(l):
    """Convert a list of numbers in string""
    \only<2-4>{s = \"{\" + str(l[0])}
    \only<3-4>{for x in l[1:]: s = s + "," + str(x)}
    \only<4>{s = s + \"\"}
    return s
\end{Piton}
\end{frame}
\end{document}
\end{verbatim}

In the previous example, the braces in the Python strings "," and \"\" are correctly interpreted (without any escape character).
5.5.3 Environments of Beamer allowed in \{Piton\} and \PitonInputFile

When \texttt{piton} is used in the class \texttt{beamer}, the following environments of Beamer are directly detected in the environments \{Piton\} (and in the listings processed by \PitonInputFile): \{actionenv\}, \{alertenv\}, \{invisibleenv\}, \{onlyenv\}, \{uncoverenv\} and \{visibleenv\}. However, there is a restriction: these environments must contain only whole lines of Python code in their body.

Here is an example:

\documentclass{beamer}
\usepackage{piton}
\begin{document}
\begin{frame}[fragile]
\begin{Piton}
def square(x):
    """Compute the square of its argument""
    \begin{uncoverenv}<2>
    return x*x
    \end{uncoverenv}
\end{Piton}
\end{frame}
\end{document}

Remark concerning the command \texttt{\alert} and the environment \{alertenv\} of Beamer

Beamer provides an easy way to change the color used by the environment \{alertenv\} (and by the command \texttt{\alert} which relies upon it) to highlight its argument. Here is an example:

\setbeamercolor{alerted text}{fg=blue}

However, when used inside an environment \{Piton\}, such tuning will probably not be the best choice because \texttt{piton} will, by design, change (most of the time) the color the different elements of text. One may prefer an environment \{alertenv\} that will change the background color for the elements to be highlighted.

Here is a code that will do that job and add a yellow background. That code uses the command \texttt{\highlight} of \texttt{lua-ul} (that extension requires also the package \texttt{luacolor}).

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

That code redefines locally the environment \{alertenv\} within the environments \{Piton\} (we recall that the command \texttt{\alert} relies upon that environment \{alertenv\}).

5.6 Footnotes in the environments of piton

If you want to put footnotes in an environment \{Piton\} or (or, more unlikely, in a listing produced by \PitonInputFile), you can use a pair \texttt{\footnotemark}–\texttt{\footnotetext}.

However, it’s also possible to extract the footnotes with the help of the package \texttt{footnote} or the package \texttt{footnotehyper}.

If \texttt{piton} is loaded with the option \texttt{footnote} (with \texttt{\usepackage[footnote]{piton}} or with \texttt{\PassOptionsToPackage}), the package \texttt{footnote} is loaded (if it is not yet loaded) and it is used to extract the footnotes.

If \texttt{piton} is loaded with the option \texttt{footnotehyper}, the package \texttt{footnotehyper} is loaded (if it is not yet loaded) and it is used to extract footnotes.
Caution: The packages \texttt{footnote} and \texttt{footnotehyper} are incompatible. The package \texttt{footnotehyper} is the successor of the package \texttt{footnote} and should be used preferentially. The package \texttt{footnote} has some drawbacks, in particular: it must be loaded after the package \texttt{xcolor} and it is not perfectly compatible with \texttt{hyperref}.

In this document, the package \texttt{piton} has been loaded with the option \texttt{footnotehyper}. For examples of notes, \cf \texttt{6.3}, \texttt{p. 18}.

## 5.7 Tabulations

Even though it’s recommended to indent the Python listings with spaces (see PEP 8), \texttt{piton} accepts the characters of tabulation (that is to say the characters U+0009) at the beginning of the lines. Each character U+0009 is replaced by \(n\) spaces. The initial value of \(n\) is \(4\) but it’s possible to change it with the key \texttt{tab-size} of \texttt{\PitonOptions}.

There exists also a key \texttt{tabs-auto-gobble} which computes the minimal value \(n\) of the number of consecutive characters U+0009 beginning each (non empty) line of the environment \{\texttt{Piton}\} and applies \texttt{gobble} with that value of \(n\) (before replacement of the tabulations by spaces, of course). Hence, that key is similar to the key \texttt{auto-gobble} but acts on U+0009 instead of U+0020 (spaces).

## 6 Examples

### 6.1 Line numbering

We remind that it’s possible to have an automatic numbering of the lines in the Python listings by using the key \texttt{line-numbers}.

By default, the numbers of the lines are composed by \texttt{piton} in an overlapping position on the left (by using internally the command \texttt{\llap} of \LaTeX{}).

In order to avoid that overlapping, it’s possible to use the option \texttt{left-margin=auto} which will insert automatically a margin adapted to the numbers of lines that will be written (that margin is larger when the numbers are greater than 10).

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

### 6.2 Formatting of the LaTeX comments

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

```
\PitonOptions{background-color=gray!10}
\SetPitonStyle{Comment.LaTeX = \hfill \normalfont\color{gray}}
\begin{Piton}
def arctan(x,n=10):
```

17
if x < 0:
    return -arctan(-x)  #> recursive call
elif x > 1:
    return pi/2 - arctan(1/x)  #> other recursive call
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)  #> recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)  #> another recursive call
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
\end{Piton}

6.3 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 5.6 p. 16. 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.
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}

\setmonofont{DejaVu Sans Mono}{Scale=0.85}
\SetPitonStyle

6.4 An example of tuning of the styles

The graphical styles have been presented in the section 4.2, p. 6.

We present now an example of tuning of these styles adapted to the documents in black and white. We use the font \textit{Deja Vu Sans Mono}\footnote{See: \url{https://dejavu-fonts.github.io}} specified by the command \texttt{\setmonofont} of \texttt{fontspec}.

That tuning uses the command \texttt{\highlight} of \texttt{lua-ult} (that package requires itself the package \texttt{luacolor}).

\setmonofont[Scale=0.85]{DejaVu Sans Mono}
\SetPitonStyle
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)
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s

6.5 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 (for technical reasons, the ! is mandatory in the signature of the environment).

\ExplSyntaxOn
\NewDocumentEnvironment { PitonExecute } { ! O { } } % the ! is mandatory
{\PyLTVerbatimEnv
\begin{pythonq}
}
{\end{pythonq}
\directlua
{tex.print("\\PitonOptions{#1}"
\text.print("\\begin{Piton}"
\text.print(pyluatex.get_last_code())
\text.print("\\end{Piton}"
\ExplSyntaxOff
This environment \texttt{PitonExecute} takes in as optional argument (between square brackets) the options of the command \texttt{PitonOptions}.
7 The styles for the different computer languages

7.1 The language Python

In piton, the default language is Python. If necessary, it’s possible to come back to the language Python with \PitonOptions{language=Python}.

<table>
<thead>
<tr>
<th>Style</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>the numbers</td>
</tr>
<tr>
<td>String.Short</td>
<td>the short strings (entre ' ou &quot;)</td>
</tr>
<tr>
<td>String.Long</td>
<td>the long strings (entre &quot;'&quot; ou &quot;&quot;&quot;&quot;) excepted the doc-strings (governed by String.Doc)</td>
</tr>
<tr>
<td>String</td>
<td>that key fixes both String.Short et String.Long</td>
</tr>
<tr>
<td>String.Doc</td>
<td>the doc-strings (only with &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 { et })；that style inherits for the styles String.Short and String.Long (accordning the kind of string where the interpolation appears)</td>
</tr>
<tr>
<td>Interpol.Inside</td>
<td>the content of the interpolations in the f-strings (that is to say the elements between { and })；if the final user has not set that key, those elements will be formatted by piton as done for any Python code.</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 et not</td>
</tr>
<tr>
<td>Name.Builtin</td>
<td>almost all the functions predefined by Python</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</td>
</tr>
<tr>
<td>Name.Class</td>
<td>the name of the Python classes defined by the user at their point of definition (with the keyword class)</td>
</tr>
<tr>
<td>Name.Function</td>
<td>the name of the Python functions defined by the user at their point of definition (with the keyword def)</td>
</tr>
<tr>
<td>UserFunction</td>
<td>the name of the Python functions previously defined by the user (the initial value of that parameter is empty and, hence, these elements are drawn, by default, in the current color, usually black)</td>
</tr>
<tr>
<td>Exception</td>
<td>les exceptions prédéfinies (ex.: SyntaxError)</td>
</tr>
<tr>
<td>InitialValues</td>
<td>the initial values (and the preceding symbol =) of the optional arguments in the definitions of functions; if the final user has not set that key, those elements will be formatted by piton as done for any Python code.</td>
</tr>
<tr>
<td>Comment</td>
<td>the comments beginning with #</td>
</tr>
<tr>
<td>Comment.LaTeX</td>
<td>the comments beginning with #&gt;, which are composed by piton as LaTeX code (merely named “LaTeX comments” in this document)</td>
</tr>
<tr>
<td>Keyword.Constant</td>
<td>True, False et None</td>
</tr>
<tr>
<td>Keyword</td>
<td>the following keywords: assert, break, case, continue, del, elif, else, except, exec, finally, for, from, global, if, import, lambda, non local, pass, raise, return, try, while, with, yield et yield from.</td>
</tr>
</tbody>
</table>
7.2 The language OCaml

It's possible to switch to the language OCaml with `\PitonOptions{language = OCaml}`.

It's also possible to set the language OCaml for an individual environment `{Piton}`:

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

<table>
<thead>
<tr>
<th>Style</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>the numbers</td>
</tr>
<tr>
<td>String.Short</td>
<td>the characters (between ')</td>
</tr>
<tr>
<td>String.Long</td>
<td>the strings, between &quot; but also the quoted-strings</td>
</tr>
<tr>
<td>Operator</td>
<td>les opérateurs, en particulier +, -, /, *, @, !=, ==, &amp;&amp;</td>
</tr>
<tr>
<td>Operator.Word</td>
<td>les opérateurs suivants : and, asr, land, lor, lsl, lxor, mod et or</td>
</tr>
<tr>
<td>Name.Builtin</td>
<td>les fonctions not, incr, decr, fst et snd</td>
</tr>
<tr>
<td>Name.Type</td>
<td>the name of a type of OCaml</td>
</tr>
<tr>
<td>Name.Field</td>
<td>the name of a field of a module</td>
</tr>
<tr>
<td>Name.Constructor</td>
<td>the name of the constructors of types (which begins by a capital)</td>
</tr>
<tr>
<td>Name.Module</td>
<td>the name of the modules</td>
</tr>
<tr>
<td>Name.Function</td>
<td>the name of the Python functions defined by the user at their point of definition (with the keyword let)</td>
</tr>
<tr>
<td>UserFunction</td>
<td>the name of the Python functions previously defined by the user</td>
</tr>
<tr>
<td></td>
<td>(the initial value of that parameter is empty and these elements are drawn in the current color, usually black)</td>
</tr>
<tr>
<td>Exception</td>
<td>the predefined exceptions (eg : End_of_File)</td>
</tr>
<tr>
<td>TypeParameter</td>
<td>the parameters of the type</td>
</tr>
<tr>
<td>Comment</td>
<td>the comments, between (* et *); these comments may be nested</td>
</tr>
<tr>
<td>Keyword.Constant</td>
<td>true et false</td>
</tr>
<tr>
<td>Keyword</td>
<td>the following keywords: assert, as, begin, class, constraint, done, downto, do, else, end, exception, external, for, function, functor, fun, if include, inherit, initializer, in, lazy, let, match, method, module, mutable, new, object, of, open, private, raise, rec, sig, struct, then, to, try, type, value, val, virtual, when, while and with</td>
</tr>
</tbody>
</table>
7.3 The language C (and C++)

It's possible to switch to the language C with \PitonOptions{language = C}.

It's also possible to set the language C for an individual environment \{Piton\}.

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

<table>
<thead>
<tr>
<th>Style</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>the numbers</td>
</tr>
<tr>
<td>String.Long</td>
<td>the strings (between &quot;)</td>
</tr>
<tr>
<td>String.Interpol</td>
<td>the elements %d, %i, %f, %c, etc. in the strings; that style inherits from the style String.Long</td>
</tr>
<tr>
<td>Operator</td>
<td>the following operators: != == &lt;&lt; &gt;&gt; - + / * % = &lt; &gt; &amp; .</td>
</tr>
<tr>
<td>Name.Type</td>
<td>the following predefined types: 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</td>
</tr>
<tr>
<td>Name.Builtin</td>
<td>the following predefined functions: printf, scanf, malloc, sizeof and alignof</td>
</tr>
<tr>
<td>Name.Class</td>
<td>le nom des classes au moment de leur définition, c'est-à-dire après le mot-clé class</td>
</tr>
<tr>
<td>Name.Function</td>
<td>the name of the Python functions defined by the user at their point of definition (with the keyword let)</td>
</tr>
<tr>
<td>UserFunction</td>
<td>the name of the Python functions previously defined by the user (the initial value of that parameter is empty and these elements are drawn in the current color, usually black)</td>
</tr>
<tr>
<td>Preproc</td>
<td>the instructions of the preprocessor (beginning par #)</td>
</tr>
<tr>
<td>Comment</td>
<td>the comments (beginning by // or between /* and */)</td>
</tr>
<tr>
<td>Comment.LaTeX</td>
<td>the comments beginning by ///&lt;/&gt; which are composed by piton as LaTeX code (merely named “LaTeX comments” in this document)</td>
</tr>
<tr>
<td>Keyword.Constant</td>
<td>default, false, NULL, nullptr and true</td>
</tr>
<tr>
<td>Keyword</td>
<td>the following keywords: 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 and while</td>
</tr>
</tbody>
</table>
8 Implementation

The development of the extension piton is done on the following GitHub depot:
https://github.com/fpantigny/piton

8.1 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 python. 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 tex.tprint to each element of that table.\(^{21}\)

Consider, for example, the following Python code:

```python
def parity(x):
    return x%2
```

The capture returned by the lpeg python against that code is the Lua table containing the following elements:

```lua
{ "\_\_\_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, " " },
{ "\{PitonStyle{Keyword}{" },
{ luatexbase.catcodetables.CatcodeTableOther, "return" },
{ "}" },
{ luatexbase.catcodetables.CatcodeTableOther, " " },
{ luatexbase.catcodetables.CatcodeTableOther, "x" },
{ "\{PitonStyle{Operator}{" },
{ luatexbase.catcodetables.CatcodeTableOther, ":" },
{ "}" },
{ "\{PitonStyle{Number}{" },
{ luatexbase.catcodetables.CatcodeTableOther, "2" },
{ "}" },
{ "\_\_\_piton_end_line:" }
```

\(^{a}\)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 \_\_\_begin_line: (since there can’t be lines breaks in the argument of a command \_\_\_begin_line:).

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

\(^{c}\)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).

\(^{21}\)Recall that 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.
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 \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 \ExplSyntaxOn)

\__piton_begin_line:{\PitonStyle{Keyword}{def}}
\{\PitonStyle{Name.Function}{parity}\}(x):\__piton_end_line:\__piton_newline:
\__piton_begin_line:␣␣␣␣\{\PitonStyle{Keyword}{return}\}
\x{\PitonStyle{Operator}{%}}{\PitonStyle{Number}{2}}\__piton_end_line:

8.2 The L3 part of the implementation

8.2.1 Declaration of the package

\langle{∗STY}\rangle
\NeedsTeXFormat{LaTeX2e}
\RequirePackage{l3keys2e}
\ProvidesExplPackage{piton}{\myfiledate}{\myfileversion}{Highlight Python codes with LPEG on LuaLaTeX}
\cs_new_protected:Npn \@@_error:n { \msg_error:nn { piton } }
\cs_new_protected:Npn \@@_warning:n { \msg_warning:nn { piton } }
\cs_new_protected:Npn \@@_error:nn { \msg_error:nnn { piton } }
\cs_new_protected:Npn \@@_error:nnn { \msg_error:nnnn { piton } }
\cs_new_protected:Npn \@@_fatal:n { \msg_fatal:nn { piton } }
\cs_new_protected:Npn \@@_fatal:nn { \msg_fatal:nnn { piton } }
\cs_new_protected:Npn \@@_msg_new:nn { \msg_new:nnn { piton } }
\@@_msg_new:nn { LuaLaTeX-mandatory }
{ LuaLaTeX-is-mandatory.\\
The-package-'piton'-requires-the-engine-LuaLaTeX.\\
\str_if_eq:VnT \c_sys_jobname_str { output } { 
{ If-you-use-Overleaf,-you-can-switch-to-LuaLaTeX-in-the-'Menu'. } 
If-you-go-on,-the-package-'piton'-won't-be-loaded.
}
\sys_if_engine_luatex:F { \msg_critical:nn { piton } { LuaLaTeX-mandatory } }
\RequirePackage { lualatexbase }
\@@_msg_new:nn { piton.lua-not-found }
{ The-file-'piton.lua'-can't-be-found.\\
The package-'piton'-won't be loaded.
}
\file_if_exist:nF { piton.lua }
{ \msg_critical:nn { piton } { piton.lua-not-found } }

The boolean \_\_\_footnotehyper_bool will indicate if the option footnotehyper is used.
\bool_new:N \_\_\_footnotehyper_bool
The boolean \_\_\_footnote_bool will indicate if the option footnote is used, but quickly, it will also be set to true if the option footnotehyper is used.
\bool_new:N \_\_\_footnote_bool

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.
\keys_define:nn { piton / package }
{
footnote .bool_gset:N = \g_@@_footnote_bool ,
footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
beamer .bool_gset:N = \g_@@_beamer_bool ,
beamer .default:n = true ,
escape-inside .code:n = \@@_error:n { key-escape-inside-deleted },
math-comments .code:n = \@@_error:n { moved-to-preamble },
comment-latex .code:n = \@@_error:n { moved-to-preamble },
unknown .code:n = \@@_error:n { Unknown-key-for-package }
}
\@@_msg_new:nn { key-escape-inside-deleted }
{
The-key-'escape-inside'-has-been-deleted.-You-must-now-use-
the-keys-'begin-escape'-and-'end-escape'-in-
\token_to_str:N \PitonOptions.\That-key-will-be-ignored.
}
\@@_msg_new:nn { moved-to-preamble }
{
The-key- '{\l_keys_key_str}'-must-now-be-used-with-
\token_to_str:N \PitonOptions' in-the-preamble-of-your-
document.\That-key-will-be-ignored.
}
\@@_msg_new:nn { Unknown-key-for-package }
{
Unknown-key.\You-have-used-the-key- '{\l_keys_key_str}'-but-the-only-keys-available-here-
are-'beamer','footnote','footnotehyper'.-Other-keys-are-available-in-
\token_to_str:N \PitonOptions.\That-key-will-be-ignored.
}

We process the options provided by the user at load-time.
\ProcessKeysOptions { piton / package }
\iftagclassloaded { beamer } \bool_gset_true:N \g_@@_beamer_bool \}{ }
\iftagpackageloaded { beamerarticle } \bool_gset_true:N \g_@@_beamer_bool \}{ }
\bool_if:NT \g_@@_beamer_bool \lua_now:n { piton_bearer = true } \}
\hook_gput_code:nnn { begindocument } \{ . }
\ifpackageloaded { xcolor } \}{ }\msg_fatal:nn { piton } { xcolor-not-loaded } \}
\@@_msg_new:nn { xcolor-not-loaded }
{
The package 'xcolor' is required by 'piton'. This error is fatal.

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.

The class beamer has its own system to extract footnotes and that's why we have nothing to do if beamer is used.

The flag \g@@_footnote_bool is raised and so, we will only have to test \g@@_footnote_bool in order to know if we have to insert an environment \savenotes.

8.2.2 Parameters and technical definitions

The following string will contain the name of the informatic language considered (the initial value is python).
In order to have a better control over the keys.

\bool_new:N \l_@@_in_PitonOptions_bool
\bool_new:N \l_@@_in_PitonInputFile_bool

The following flag will be raised in the \AtBeginDocument.
\bool_new:N \g_@@_in_document_bool

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 contain 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 \textit{n}, then no line break can occur within the first \textit{n} lines or the last \textit{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.
\clist_new:N \l_@@_bg_color_clist

The package piton will also detect the lines of code which correspond to the user input in a Python console, that is to say the lines of code beginning with >>> and .... It’s possible, with the key prompt-background-color, to require a background for these lines of code (and the other lines of code will have the standard background color specified by background-color).
\tl_new:N \l_@@_prompt_bg_color_tl

The following parameters correspond to the keys begin-range and end-range of the command \PitonInputFile.
\str_new:N \l_@@_begin_range_str
\str_new:N \l_@@_end_range_str

The argument of \PitonInputFile.
\str_new:N \l_@@_file_name_str

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

The following boolean corresponds to the key show-spaces.
\bool_new:N \l_@@_show_spaces_bool

The following booleans correspond to the keys break-lines and indent-broken-lines.
\bool_new:N \l_@@_break_lines_in_Piton_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 \break-lines-in-piton\%.
\bool_new:N \l_@@_break_lines_in_piton_bool

The following dimension will be the width of the listing constructed by \Piton or \PitonInputFile\%.
• If the user uses the key \width of \PitonOptions with a numerical value, that value will be stored in \l_@@_width_dim.
• If the user uses the key \width with the special value \texttt{min}, the dimension \l_@@_width_dim will, in the second run, be computed from the value of \l_@@_line_width_dim stored in the aux file (computed during the first run the maximal width of the lines of the listing). During the first run, \l_@@_width_line_dim will be set equal to \linewidth.
• Elsewhere, \l_@@_width_dim will be set at the beginning of the listing (in \@@_pre_env\%) equal to the current value of \linewidth.
\dim_new:N \l_@@_width_dim

We will also use another dimension called \l_@@_line_width_dim. That will the width of the actual lines of code. That dimension may be lower than the whole \l_@@_width_dim because we have to take into account the value of \l_@@_left_margin_dim (for the numbers of lines when line-numbers is in force) and another small margin when a background color is used (with the key \background-color\%).
\dim_new:N \l_@@_line_width_dim

The following flag will be raised with the key \width is used with the special value \texttt{min}.
\bool_new:N \l_@@_width_min_bool

If the key \width is used with the special value \texttt{min}, we will compute the maximal width of the lines of an environment \{Piton\} in \g_@@_tmp_width_dim because we need it for the case of the key \width is used with the spacial value \texttt{min}. We need a global variable because, when the key \footnote is in force, each line when be composed in an environment \{savenotes\} and we need to exit our \g_@@_tmp_width_dim from that environment.
\dim_new:N \g_@@_tmp_width_dim

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

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

The following dimension corresponds to the key \numbers-sep of \PitonOptions\%.
\dim_new:N \l_@@_numbers_sep_dim
\dim_set:Nn \l_@@_numbers_sep_dim { 0.7 \text{ em} }\%

The tabulators will be replaced by the content of the following token list.
\cs_new_protected:Npn \@@_set_tab_tl:n #1 \%
\tl_clear:N \l_@@_tab_tl
\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 }

The following command \@@_beamer_command:n executes the argument corresponding to its argument but also stores it in \l_@@_beamer_command_str. That string is used only in the error message “cr-not-allowed” raised when there is a carriage return in the mandatory argument of that command.

\cs_new_protected:Npn \@@_beamer_command:n #1
{ \str_set:Nn \l_@@_beamer_command_str { #1 } \use:c { #1 } }

In the environment \{Piton\}, the command \label will be linked to the following command.

\cs_new_protected:Npn \@@_label:n #1
{ \bool_if:NTF \l_@@_line_numbers_bool { \@bsphack \protected@write \@auxout { \string \newlabel { #1 } { \int_eval:n { \g_@@_visual_line_int + 1 } } \thepage } \@esphack } { \@@_error:n { label~with~lines~numbers } } }

Remember that the content of a line is typeset in a box before the composition of the potential number of line.

\{ \int_eval:n { \g_@@_visual_line_int + 1 } \} \thepage 
\@esphack
{ \@@_error:n { label-with-lines-numbers } }

The following commands corresponds to the keys marker/beginning and marker/end. The values of that keys are functions that will be applied to the “range” specified by the final user in an individual \PitonInputFile. They will construct the markers used to find textually in the external file loaded by piton the part which must be included (and formatted).

\cs_new_protected:Npn \@@_marker_beginning:n #1 { }
\cs_new_protected:Npn \@@_marker_end:n #1 { }

The following commands are a easy way to insert safely braces ({ and }) in the TeX flow.

\cs_new_protected:Npn \@@_open_brace: { \directlua { piton.open_brace() } }
\cs_new_protected:Npn \@@_close_brace: { \directlua { piton.close_brace() } }

The following token list will be evaluated at the beginning of \@@_begin_line:... \@@_end_line: and cleared at the end. It will be used by LPEG acting between the lines of the Python code in order to add instructions to be executed at the beginning of the line.

\tl_new:N \g_@@_begin_line_hook_tl
For example, the LPEG Prompt will trigger the following command which will insert an instruction in the hook \g_@@_begin_line_hook to specify that a background must be inserted to the current line of code.

```
\cs_new_protected:Npn \@@_prompt:
\{ 
  \tl_gset:Nn \g_@@_begin_line_hook_tl 
  \{ 
    \tl_if_empty:NF \l_@@_prompt_bg_color_tl % added 2023-04-24 
    \clist_set:NV \l_@@_bg_color_clist \l_@@_prompt_bg_color_tl 
  } 
\}
```

8.2.3 Treatment of a line of code

```
\cs_new_protected:Npn \@@_replace_spaces:n #1
\{ 
  \tl_set:Nn \l_tmpa_tl { #1 }
  \bool_if:NTF \l_@@_show_spaces_bool
  \{ 
    \regex_replace_all:nnN { \x20 } { \c { \@@_breakable_space: } } \l_tmpa_tl 
  % U+2423
  \}
  \bool_if:NT \l_@@_break_lines_in_Piton_bool
  \{ 
    \regex_replace_all:nnN { \x20 } { \c { \@@_breakable_space: } } \l_tmpa_tl 
  \}
  \cs_generate_variant:Nn \@@_replace_spaces:n { x }
```

In the contents provided by Lua, each line of the Python code will be surrounded by \@@_begin_line: and \@@_end_line: . \@@_begin_line: is a LaTeX command that we will define now but \@@_end_line: is only a syntactic marker that has no definition.

```
\cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
\{ 
  \group_begin:
  \g_@@_begin_line_hook_tl
  \int_gzero:N \g_@@_indentation_int
\}
```

First, we will put in the coffin \l_tmpa_coffin the actual content of a line of the code (without the potential number of line).

Be careful: There is curryfication in the following code.

```
\bool_if:NTF \l_@@_width_min_bool
\{ 
  \@@_put_in_coffin_ii:n \@@_put_in_coffin_i:n
  \language = -1
  \raggedright
  \strut
  \@@_replace_spaces:n \{ #1 \}
  \strut \hfil 
\}
```

Now, we add the potential number of line, the potential left margin and the potential background.

```
\hbox_set:Nn \l_tmpa_box
```
If there is a background, we must remind that there is a left margin of 0.5 em for the background...

In the general case (which is also the simpler), the key \texttt{width} is not used, or (if used) it is not used with the special value \texttt{min}. In that case, the content of a line of code is composed in a vertical coffin with a width equal to \texttt{l_@@_line_width_dim}. That coffin may, eventually, contain several lines when the key \texttt{broken-lines-in-Piton} (or \texttt{broken-lines}) is used.

That command takes in its argument by curryfication.

The second case is the case when the key \texttt{width} is used with the special value \texttt{min}. 
First, we compute the natural width of the line of code because we have to compute the natural width of the whole listing (and it will be written on the aux file in the variable \l_@@_width_dim).

\hbox_set:Nn \l_tmpa_box { #1 }

Now, you can actualize the value of \g_@@_tmp_width_dim (it will be used to write on the aux file the natural width of the environment).

\dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_tmp_width_dim
{ \dim_gset:Nn \g_@@_tmp_width_dim { \box_wd:N \l_tmpa_box } }

\hcoffin_set:Nn \l_tmpa_coffin
{ \hbox_to_wd:nn \l_@@_line_width_dim
We unpack the block in order to free the potential \hfill springs present in the LaTeX comments (cf. section 6.2, p. 17).

\hbox_unpack:N \l_tmpa_box \hfil
}

The command \@@_color:N will take in as argument a reference to a comma-separated list of colors. A color will be picked by using the value of \g_@@_line_int (modulo the number of colors in the list).

\cs_set_protected:Npn \@@_color:N #1
{ \int_set:Nn \l_tmpa_int { \clist_count:N #1 }
\int_set:Nn \l_tmpb_int { \int_mod:nn \g_@@_line_int \l_tmpa_int + 1 }
\tl_set:Nx \l_tmpa_tl { \clist_item:Nn #1 \l_tmpb_int }
\tl_if_eq:NnTF \l_tmpa_tl { none }
{ \dim_zero:N \l_@@_width_dim }
{ \exp_args:NV \@@_color_i:n \l_tmpa_tl }
}

The following command \@@_color:n will accept both the instruction \@@_color:n \{ red!15 \} and the instruction \@@_color:n \{ [rgb]{0.9,0.9,0} \}.

\cs_set_protected:Npn \@@_color:n { #1 }
{ \tl_if_head_eq_meaning:nNTF { #1 } \[ }
{ \tl_set:Nn \l_tmpa_tl { #1 }
\tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
\exp_last_unbraced:NV \color \l_tmpa_tl
}
\color { #1 }
}

The command \@@_newline: will set \l_@@_width_dim to zero, the colored rectangle will be drawn with zero width and, thus, it will be a mere strut (and we need that strut).

\int_gincr:N \g_@@_line_int
\int_compare:nNnT \g_@@_line_int > \l_@@_splittable_int
{ \int_compare:nNnT \l_@@_nb_lines_int - \g_@@_line_int > \l_@@_splittable_int
{ \egroup
\bool_if:NT \g_@@_footnote_bool { \end { savenotes } }
\par \mode_leave_vertical: % \newline
\bool_if:NT \g_@@_footnote_bool { \begin { savenotes } }
\vtop \bgroup

34
8.2.4 PitonOptions

\bool_new:N \l_@@_line_numbers_bool
\bool_new:N \l_@@_skip_empty_lines_bool
\bool_new:N \l_@@_line_numbers_absolute_bool
\bool_new:N \l_@@_label_empty_lines_bool
\int_new:N \l_@@_number_lines_start_int
\bool_new:N \l_@@_resume_bool

\keys_define:nn { PitonOptions / marker }
{ beginning .code:n = \cs_set:Nn \@@_marker_beginning:n { #1 } ,
  beginning .value_required:n = true ,
  end .code:n = \cs_set:Nn \@@_marker_end:n { #1 } ,
  end .value_required:n = true ,
  include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
  include-lines .default:n = true ,
  unknown .code:n = \@@_error:n { Unknown-key-for-marker } }

\keys_define:nn { PitonOptions / line-numbers }
{ true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
  false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
  start .code:n = \bool_if:NTF \l_@@_in_PitonOptions_bool
    { Invalid-key } }


\bool_set_true:N \l_@@_line_numbers_bool
\int_set:Nn \l_@@_number_lines_start_int { #1 }

start .value_required:n = true ,
skip-empty-lines .code:n =
\bool_if:NF \l_@@_in_PitonOptions_bool
{ \bool_set_true:N \l_@@_line_numbers_bool }
\str_if_eq:nnTF { #1 } { false }
{ \bool_set_false:N \l_@@_skip_empty_lines_bool }
{ \bool_set_true:N \l_@@_skip_empty_lines_bool },
skip-empty-lines .default:n = true ,

label-empty-lines .code:n =
\bool_if:NF \l_@@_in_PitonOptions_bool
{ \bool_set_true:N \l_@@_line_numbers_bool }
\str_if_eq:nnTF { #1 } { false }
{ \bool_set_false:N \l_@@_label_empty_lines_bool }
{ \bool_set_true:N \l_@@_label_empty_lines_bool },
label-empty-lines .default:n = true ,

absolute .code:n =
\bool_if:NTF \l_@@_in_PitonOptions_bool
{ \bool_set_true:N \l_@@_line_numbers_absolute_bool }
{ \bool_set_true:N \l_@@_line_numbers_bool }
\bool_if:NT \l_@@_in_PitonInputFile_bool
{ \bool_set_true:N \l_@@_line_numbers_absolute_bool }
\bool_set_false:N \l_@@_skip_empty_lines_bool
{ \bool_lazy_or:nnF \l_@@_in_PitonInputFile_bool
\l_@@_in_PitonOptions_bool
\l_@@_in_PitonOptions_bool
{ \@@_error:n { Invalid-key } } },
absolute .value_forbidden:n = true ,

resume .code:n =
\bool_set_true:N \l_@@_resume_bool
\bool_if:NF \l_@@_in_PitonOptions_bool
{ \bool_set_true:N \l_@@_line_numbers_bool }
\bool_set_true:N \l_@@_line_numbers_bool
{ \bool_set_true:N \l_@@_line_numbers_absolute_bool }
\bool_set_true:N \l_@@_line_numbers_absolute_bool
\bool_set_false:N \l_@@_skip_empty_lines_bool
{ \@@_error:n { Invalid-key } } ,
resume .value_forbidden:n = true ,

sep .dim_set:N = \l_@@_numbers_sep_dim ,
sep .value_required:n = true ,
unknown .code:n = \@@_error:n { Unknown-key-for-line-numbers }
}

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 }

First, we put keys that should be available only in the preamble.
begin-escape .code:n =
\lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
begin-escape .value_required:n = true ,
begin-escape .usage:n = preamble ,
end-escape .code:n =
\lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
end-escape .value_required:n = true ,
end-escape .usage:n = preamble ,
begin-escape-math .code:n =\lua_now:e { piton.begin_escape_math = "\lua_escape:n(#1)" },
begin-escape-math .value_required:n = true,
begin-escape-math .usage:n = preamble,
end-escape-math .code:n =\lua_now:e { piton.end_escape_math = "\lua_escape:n(#1)" },
end-escape-math .value_required:n = true,
end-escape-math .usage:n = preamble,

comment-latex .code:n = \lua_now:n { comment_latex = "#1" },
comment-latex .value_required:n = true,
comment-latex .usage:n = preamble,

math-comments .bool_set:N = \g_@@_math_comments_bool,
math-comments .default:n = true,
math-comments .usage:n = preamble,

Now, general keys.

language .code:n =\str_set:Nx \l_@@_language_str { \str_lowercase:n { #1 } },
language .value_required:n = true,
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,
tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 },
tabs-auto-gobble .value_forbidden:n = true,

marker .code:n =
\bool_lazy_or:nnTF
\l_@@_in_PitonInputFile_bool
\l_@@_in_PitonOptions_bool
{ \keys_set:nn { PitonOptions / marker } { #1 } }
{ \@@_error:n { Invalid~key } },
marker .value_required:n = true,

line-numbers .code:n =
\keys_set:nn { PitonOptions / line-numbers } { #1 },
line-numbers .default:n = true,

splittable .int_set:N = \l_@@_splittable_int,
splittable .default:n = 1,
background-color .clist_set:N = \l_@@_bg_color_clist,
background-color .value_required:n = true,
prompt-background-color .tl_set:N = \l_@@_prompt_bg_color_tl,
prompt-background-color .value_required:n = true,

width .code:n =\str_if_eq:nnTF { #1 } { min }
{ \bool_set_true:N \l_@@_width_min_bool
 \dim_zero:N \l_@@_width_dim
 }
{ \bool_set_false:N \l_@@_width_min_bool
 \dim_set:Nn \l_@@_width_dim { #1 } }
width .value_required: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 }
    \bool_set_false:N \l_@@_left_margin_auto_bool
},
left-margin .value_required:n = true,

% deprecated

\bool_set_true:N \l_@@_line_numbers_bool
8.2.5 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).

\int_new:N \g_@@_visual_line_int
\cs_new_protected:Npn \@@_print_number: 
\hbox_overlap_left:n 
\color { gray } \footnotesize \int_to_arabic:n \g_@@_visual_line_int \skip_horizontal:N \l_@@_numbers_sep_dim

8.2.6 The command to write on the aux file

\cs_new_protected:Npn \@@_write_aux: 
\tl_if_empty:NF \g_@@_aux_tl { \ExplSyntaxOn } \tl_gset:cn { c_@@_ \int_use:N \g_@@_env_int _ tl } \exp_not:V \g_@@_aux_tl \tl_gclear:N \g_@@_aux_tl
The following macro will be used only when the key `width` is used with the special value `min`.

```latex
\cs_new_protected:Npn \@@_width_to_aux:
\{
  \tl_gput_right:Nx \g_@@_aux_tl
  { \dim_set:Nn \l_@@_line_width_dim
  \{ \dim_eval:n \{ \g_@@_tmp_width_dim \} \}
  }
\}
```

### 8.2.7 The main commands and environments for the final user

```latex
\NewDocumentCommand { \piton } { } { \peek_meaning:NTF \bgroup \@@_piton_standard \@@_piton_verbatim }
\NewDocumentCommand { \@@_piton_standard } { m } {\group_begin:, \ttfamily
\automatichyphenmode = 1
\cs_set_eq:NN \\ \c_backslash_str
\cs_set_eq:NN \% \c_percent_str
\cs_set_eq:NN \{ \c_left_brace_str
\cs_set_eq:NN \} \c_right_brace_str
\cs_set_eq:NN \$ \c_dollar_str
\cs_set_eq:cN { ~ } \space
\cs_set_protected:Npn \@@_begin_line: { }
\cs_set_protected:Npn \@@_end_line: { }
\tl_set:Nx \l_tmpa_tl
\{ \lua_now:e
  \{ \l_@@_language_str', token.scan_string() \}
  \{ \#1 \}
\}
\bool_if:NTF \l_@@_show_spaces_bool
  \{ \regex_replace_all:nnN { \x20 } { \x20 } \l_tmpa_tl \}
\l_tmpa_tl
\group_end:
\}
\NewDocumentCommand { \@@_piton_verbatim } { v } { \group_begin:, \ttfamily
\automatichyphenmode = 1
\cs_set_protected:Npn \@@_begin_line: { }
\cs_set_protected:Npn \@@_end_line: { }
\tl_set:Nx \l_tmipa_t1
\{ \lua_now:e
  \{ \l_@@_language_str', token.scan_string() \}
  \{ \#1 \}
\}
\bool_if:NTF \l_@@_show_spaces_bool
  \{ \regex_replace_all:nnN { \x20 } { \x20 } \l_tmipa_t1 \}
\l_tmipa_t1
\group_end:
\}
\NewDocumentCommand { \@@_piton_verbatim } { v } { \group_begin:, \ttfamily
\automatichyphenmode = 1
\cs_set_protected:Npn \@@_begin_line: { }
\cs_set_protected:Npn \@@_end_line: { }
\tl_set:Nx \l_tmipa_t1
\{ \lua_now:e
  \{ \l_@@_language_str', token.scan_string() \}
  \{ \#1 \}
\}
\bool_if:NTF \l_@@_show_spaces_bool
  \{ \regex_replace_all:nnN { \x20 } { \x20 } \l_tmipa_t1 \}
\l_tmipa_t1
```

The following tuning of LuaTeX in order to avoid all break of lines on the hyphens.

```latex
\automatichyphenmode = 1
\cs_set_protected:Npn \@@_begin_line: { }
\cs_set_protected:Npn \@@_end_line: { }
\tl_set:Nx \l_tmipa_t1
\{ \lua_now:e
  \{ \l_@@_language_str', token.scan_string() \}
  \{ \#1 \}
\}
```

The following code replaces the characters U+0020 (spaces) by characters U+0020 of catcode 10: thus, they become breakable by an end of line.

```latex
\bool_if:NT \l_@@_break_lines_in_piton_bool
  \{ \regex_replace_all:nnN { \x20 } { \x20 } \l_tmipa_t1 \}
\l_tmipa_t1
\group_end:
\}
```

The following code replaces the characters U+0020 (spaces) by characters U+0020 of catcode 10: thus, they become breakable by an end of line.

```latex
\bool_if:NT \l_@@_break_lines_in_piton_bool
  \{ \regex_replace_all:nnN { \x20 } { \x20 } \l_tmipa_t1 \}
\l_tmipa_t1
\group_end:
```

```latex
\l_tmipa_t1
```

40
The following command is not a user command. It will be used when we will have to “rescan” some chunks of Python code. For example, it will be the initial value of the Piton style `InitialValues` (the default values of the arguments of a Python function).

```latex
\cs_new_protected:Npn \@@_piton:n #1
\{
  \group_begin:
  \cs_set_protected:Npn \@@_begin_line: { }
  \cs_set_protected:Npn \@@_end_line: { }
  \bool_lazy_or:nnTF
    \l_@@_break_lines_in_piton_bool
    \l_@@_break_lines_in_Piton_bool
    \{
      \tl_set:Nx \l_tmpa_tl
      \{
        \lua_now:e
        \{ piton.ParseTer('\l_@@_language_str',token.scan_string()) \}
        \{ #1 \}
      \}
    \}
  \{
    \tl_set:Nx \l_tmpa_tl
    \{
      \lua_now:e
      \{ piton.Parse('\l_@@_language_str',token.scan_string()) \}
      \{ #1 \}
    \}
  \}
  \bool_if:NT \l_@@_show_spaces_bool
    \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl % U+2423
  \l_tmpa_tl
  \group_end:
\}
```

The following command is similar to the previous one but raise a fatal error if its argument contains a carriage return.

```latex
\cs_new_protected:Npn \@@_piton_no_cr:n #1
\{
  \group_begin:
  \cs_set_protected:Npn \@@_begin_line: { }
  \cs_set_protected:Npn \@@_end_line: { }
  \cs_set_protected:Npn \@@_newline:
    \msg_fatal:nn { piton } { cr~not~allowed }
  \bool_lazy_or:nnTF
    \l_@@_break_lines_in_piton_bool
    \l_@@_break_lines_in_Piton_bool
    \{
      \tl_set:Nx \l_tmpa_tl
      \{
        \lua_now:e
        \{ piton.ParseTer('\l_@@_language_str',token.scan_string()) \}
        \{ #1 \}
      \}
    \}
  \{
    \tl_set:Nx \l_tmpa_tl
    \{
      \lua_now:e
      \{ piton.Parse('\l_@@_language_str',token.scan_string()) \}
      \{ #1 \}
    \}
  \}
  \bool_if:NT \l_@@_show_spaces_bool
    \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl % U+2423
  \l_tmpa_tl
  \group_end:
\}
```
Despite its name, \texttt{\_\_\_pre\_env:} will be used both in \texttt{PitonInputFile} and in the environments such as \texttt{\{Piton\}}.

\begin{verbatim}
\cs_new:Npn \_\_\_pre\_env:
{
\automatichyphenmode = 1
\int_gincr:N \_\_\_env_int
\tl_gclear:N \_\_\_aux_tl
\dim_compare:nNnT \_\_\_width_dim = \c_zero_dim
\cs_set_eq:NN \_\_\_left_margin_dim \_\_\_width_dim \linenumber
}
\end{verbatim}

We read the information written on the \texttt{aux} file by previous run (when the key \texttt{width} is used with the special value \texttt{min}). At this time, the only potential information written on the \texttt{aux} file is the value of \texttt{\_\_\_line\_width\_dim} when the key \texttt{width} has been used with the special value \texttt{min}).

\begin{verbatim}
\cs_if_exist_use:c { c_\_\_ \int_use:N \_\_\_env_int \_tl }
\bool_if:NF \_\_\_resume_bool { \int_gzero:N \_\_\_visual_line_int }
\dim_gzero:N \_\_\_tmp_width_dim
\int_gzero:N \_\_\_line_int
\dim_zero:N \parindent
\dim_zero:N \lineskip
\cs_set_eq:NN \label \_\_\_label:n
\end{verbatim}

If the final user has used both \texttt{left-margin=auto} and \texttt{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 \texttt{left-margin}. The first argument of the following function is the name of the Lua function that will be applied to the second argument in order to count the number of lines.

\begin{verbatim}
\cs_new_protected:Npn \_\_\_compute\_left\_margin:nn #1 #2
{
\bool_lazy_and:nnT \_\_\_left_margin_auto_bool \_\_\_line_numbers_bool
{
\hbox_set:Nn \_\_\_tmpa_box
{
{ \footnotesize
\lua_now:n
{ \g_\_\_skip_empty_lines_bool
{ \texttt{piton}.#1(\texttt{token.scan_argument}()) }
{ #2 }
\int_to_arabic:n
{ \g_\_\_visual_line_int + \_\_\_nb_non_empty_lines_int }
}
{ \int_to_arabic:n
{ \g_\_\_visual_line_int + \_\_\_nb_lines_int }
}
\dim_set:Nn \_\_\_left_margin_dim
{ \box_wd:N \_\_\_tmpa_box + \_\_\_numbers_sep_dim + 0.1 \text{em} }
}
}
\end{verbatim}

Whereas \texttt{\_\_\_with\_dim} is the width of the environment, \texttt{\_\_\_line\_width\_dim} is the width of the lines of code without the potential margins for the numbers of lines and the background.
Depending on the case, you have to compute \texttt{\_@@_line_width_dim} from \texttt{\_@_width_dim} or we have to do the opposite.

\begin{verbatim}
\cs_new_protected:Npn \@@_compute_width:
{ \dim_compare:nNnTF \l_@@_line_width_dim = \c_zero_dim
  { \dim_set_eq:NN \l_@@_line_width_dim \l_@@_width_dim
    \clist_if_empty:NTF \l_@@_bg_color_clist
      { \dim_sub:Nn \l_@@_line_width_dim \l_@@_left_margin_dim } }
\end{verbatim}

If there is no background, we only subtract the left margin.
\begin{verbatim}
\dim_sub:Nn \l_@@_line_width_dim \l_@@_left_margin_dim \}
\end{verbatim}

If there is a background, we subtract 0.5 em for the margin on the right.
\begin{verbatim}
\dim_sub:Nn \l_@@_line_width_dim { 0.5 em }\}
\end{verbatim}

And we subtract also for the left margin. If the key \texttt{left-margin} has been used (with a numerical value or with the special value \texttt{min}), \texttt{\_@@_left_margin_dim} has a non-zero value\textsuperscript{22} and we use that value. Elsewhere, we use a value of 0.5 em.
\begin{verbatim}
\dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
  { \dim_sub:Nn \l_@@_line_width_dim { 0.5 em } }\}
\end{verbatim}

If \texttt{\_@@_line_width_dim} has yet a non-empty value, that means that it has been read on the \texttt{aux} file: it has been written on a previous run because the key \texttt{width} is used with the special value \texttt{min}). We compute now the width of the environment by computations opposite to the preceding ones.
\begin{verbatim}
\dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
  { \dim_add:Nn \l_@@_line_width_dim { 0.5 em } }\}
\end{verbatim}

\NewDocumentCommand { \NewPitonEnvironment } { m m m m }
{ We construct a TeX macro which will catch as argument all the tokens until \texttt{\end{name_env}} with, in that \texttt{\end{name_env}}, the catcodes of \texttt{,}, \texttt{\{ and \} equal to 12 ("other"). The latter explains why the definition of that function is a bit complicated.
\begin{verbatim}
\use:x
{ \cs_set_protected:Npn
  \use:c { _@@_collect_ #1 :w }
  \c_backslash_str end \c_left_brace_str #1 \c_right_brace_str
} \group_end:
\end{verbatim}

We count with Lua the number of lines of the argument. The result will be stored by Lua in \texttt{\l_@@_nb_lines_int}. That information will be used to allow or disallow page breaks.
\begin{verbatim}
\lua_now:n { \texttt{piton.CountLines(token.scan_argument())} } \}
\end{verbatim}

\textsuperscript{22}If the key \texttt{left-margin} has been used with the special value \texttt{min}, the actual value of \texttt{\_@@_left_margin_dim} has yet been computed when we use the current command.
The first argument of the following function is the name of the Lua function that will be applied to the second argument in order to count the number of lines.

\@_compute_left_margin:nn \{ CountNonEmptyLines \} { ##1 }
\@_compute_width:
\ttfamily
\dim_zero:N \parskip % added 2023/07/06
\g_@@_footnote_bool
is raised when the package \texttt{piton} has been load with the key \texttt{footnote} or the key \texttt{footnotehyper}.
\bool_if:NT \g_@@_footnote_bool \{ \begin { savenotes } \vtop \bgroup
\lua_now:e
{ \l_@@_language_str ,
\int_use:N \l_@@_gobble_int ,
token.scan_argument() }
\vspace { 2.5 pt }
\egroup \bool_if:NT \g_@@_footnote_bool \{ \end { savenotes } }
\bool_if:NT \l_@@_width_min_bool \@@_width_to_aux:
The following \texttt{\end{#1}} is only for the stack of environments of \LaTeX.
\end{#1}
\@@_write_aux:
\}
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\\}
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
\}
\bool_if:NTF \g_@@_beamer_bool
\NewPitonEnvironment { Piton } { d < > O { } }
\AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
\end{#1} \{ #1 \}
\@@_write_aux:
\)
The code of the command \texttt{\PitonInputFile} is somewhat similar to the code of the environment \{\texttt{Piton}\}. In fact, it’s simpler because there isn’t the problem of catching the content of the environment in a verbatim mode.

\begin{verbatim}
\NewDocumentCommand { \PitonInputFile } { d < > O { } m }
{ \file_if_exist:nTF { #3 }
  { \@@_input_file:nnn { #1 } { #2 } { #3 } }
  { \msg_error:nnn { piton } { Unknown~file } { #3 } }
}
\cs_new_protected:Npn \@@_input_file:nnn #1 #2 #3
{ \str_set:Nn \l_@@_file_name_str { #3 }

We recall that, if we are in Beamer, the command \texttt{\PitonInputFile} is “overlay-aware” and that’s why there is an optional argument between angular brackets (< and >).

\begin{verbatim}
\tl_if_novalue:nF { #1 }
{ \bool_if:NTF \g_@@_beamer_bool
  { \begin { uncoverenv } < #1 > }
  { \@@_error:n { overlay~without~beamer } }
}
\group_begin:
\int_zero_new:N \l_@@_first_line_int
\int_zero_new:N \l_@@_last_line_int
\int_set_eq:NN \l_@@_last_line_int \c_max_int
\bool_set_true:N \l_@@_in_PitonInputFile_bool
\keys_set:nn { PitonOptions } { #2 }
\bool_if:nTF
{ \int_compare_p:nNn \l_@@_first_line_int > 0
  \| \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
}
{ \str_if_empty:nF \l_@@_begin_range_str }
\bool_if:NTF
{ \int_compare_p:nNn \l_@@_first_line_int > 0
  \| \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
}
{ \str_if_empty:nF \l_@@_begin_range_str }
\bool_lazy_or:nnT
{ \l_@@_marker_include_lines_bool }
{ ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
\end{verbatim}
\end{verbatim}
The following case arise when the code line-numbers/absolute is in force without the use of a marked range.

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.

The first argument of the following function is the name of the Lua function that will be applied to the second argument in order to count the number of lines.

We recall that, if we are in Beamer, the command \PitonInputFile is “overlay-aware” and that’s why we close now an environment {uncoverenv} that we have opened at the beginning of the command.

The following command computes the values of \l@@@first_line_int and \l@@@last_line_int when \PitonInputFile is used with textual markers.

We store the markers in L3 strings (str) in order to do safely the following replacement of \\#.

We replace the sequences \# which may be present in the prefixes (and, more unlikely, suffixes) added to the markers by the functions \@@marker_beginning:n and \@@marker_end:n

\exp_args:NnV \regex_replace_all:nnN { \\# } \c_hash_str \l_tmpa_str
\exp_args:NnV \regex_replace_all:nnN { \\# } \c_hash_str \l_tmpb_str
\lua_now:e
\{ piton.ComputeRange
   ( \l_tmpa_str , \l_tmpb_str , \l@@_file_name_str )
\}

8.2.8 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 in its argument by curryfication.
\NewDocumentCommand { \SetPitonStyle } { } { \keys_set:nn { piton / Styles } }
\cs_new_protected:Npn \@@_math_scantokens:n #1
   { \normalfont \scantextokens { $#1$ } }
\clist_new:N \g_@@_style_clist
\clist_set:Nn \g_@@_styles_clist
\Comment , \Comment.LaTeX , \Exception , \FormattingType , \Identifier , \InitialValues , \Interpol.Inside , \Keyword , \Keyword.Constant , \Name.Builtin , \Name.Class , \Name.Constructor , \Name.Decorator , \Name.Field , \Name.Function , \Name.Module , \Name.Namespace , \Name.Type , \Number , \Operator , \Operator.Word , \Preproc , \Prompt , \String.Doc , \String.Interpol , \String.Long , \String.Short , \TypeParameter , \UserFunction 
\clist_map_inline:Nn \g_@@_styles_clist
\keys_define:nn { piton / Styles }
   { #1 .tl_set:c = pitonStyle #1 , #1 .value_required:n = true }
\keys_define:nn { piton / Styles }
   {
We add the word `String` to the list of the styles because we will use that list in the error message for an unknown key in `\SetPitonStyle`.

Of course, we sort that clist.

The initial styles are inspired by the style “manni” of Pygments.
The last styles `ParseAgain.noCR` and `ParseAgain` should be considered as “internal style” (not available for the final user). However, maybe we will change that and document these styles for the final user (why not?).

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).

```latex
\bool_if:NT \g_@@_math_comments_bool { \SetPitonStyle { Comment.Math } }
```

### 8.2.10 Highlighting some identifiers

```latex
\cs_new_protected:Npn \@@_identifier:n #1
\cs_if_exist_use:c { PitonIdentifier _ \l_@@_language_str _ #1 } { #1 }
```

```latex
\keys_define:nn { PitonOptions }
\keys_define:nn { Piton / identifiers }
\keys_set:nn { Piton / identifiers } { names .clist_set:N = \l_@@_identifiers_names_tl ,
\tl_set:N = \l_@@_style_tl ,
}
```

```latex
\cs_new_protected:Npn \@@_set_identifiers:n #1
\clist_clear_new:N \l_@@_identifiers_names_tl
\tl_clear_new:N \l_@@_style_tl
\keys_set:nn { Piton / identifiers } { #1 }
\clist_map_inline:Nn \l_@@_identifiers_names_tl
\tl_set_eq:cN { PitonIdentifier _ \l_@@_language_str _ ##1 }
\l_@@_style_tl
```

In particular, we have an highlighting of the identifiers which are the names of Python functions previously defined by the user. Indeed, when a Python function is defined, the style `Name.Function.Internal` is applied to that name. We define now that style (you define it directly and you short-cut the function `\SetPitonStyle`).

```latex
\cs_gset_protected:cpn { PitonIdentifier _ \l_@@_language_str _ #1 }
\seq_if_exist:cF { g_@@_functions _ \l_@@_language_str _ seq }
\seq_new:c { g_@@_functions _ \l_@@_language_str _ seq }
\seq_gput_right:cn { g_@@_functions _ \l_@@_language_str _ seq } { #1 }
```

First, the element is composed in the TeX flow with the style `Name.Function` which is provided to the final user.

```latex
\PitonStyle { Name.Function } { #1 }
```

Now, we specify that the name of the new Python function is a known identifier that will be formatted with the Piton style `UserFunction`. Of course, here the affectionation is global because we have to exit many groups and even the environments `{Piton}`).

```latex
\PitonStyle { UserFunction } { #1 }
```

Now, we put the name of that new user function in the dedicated sequence (specific of the current language). That sequence will be used only by `\PitonClearUserFunctions`.

```latex
\seq_if_exist:cF { g_@@_functions _ \l_@@_language_str _ seq }
\seq_new:c { g_@@_functions _ \l_@@_language_str _ seq }
\seq_gput_right:cn { g_@@_functions _ \l_@@_language_str _ seq } { #1 }
```
\NewDocumentCommand \PitonClearUserFunctions { ! O { \l_@@_language_str } } { \seq_if_exist:cT { g_@@_functions _ #1 _ seq } { \seq_map_inline:cn { g_@@_functions _ #1 _ seq } { \cs_undefine:c { PitonIdentifier _ #1 _ ##1} } \seq_gclear:c { g_@@_functions _ #1 _ seq } } }

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

8.2.12 The error messages of the package
\@@_msg_new:nn { Unknown~key~for~SetPitonStyle } { The-style-\l_keys_key_str'-is-unknown.\par This-key-will-be-ignored. \par The-available-styles-are-(in-alphabetic-order):-\par \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
}
\@@_msg_new:nn { Invalid~key } { Wrong-use-of-key.\par You-can't-use-the-key-\l_keys_key_str'-here.\par That-key-will-be-ignored. }
\@@_msg_new:nn { Unknown~key~for~line-numbers } { Unknown-key. \par The-key-'line-numbers / \l_keys_key_str'-is-unknown.\par The-available-keys-of-the-family-'line-numbers'-are-(in- alphabetic-order):-\par absolute,-false,-label-empty-lines,-resume,-skip-empty-lines,- sep,-start-and-true.\par That-key-will-be-ignored. }
\@@_msg_new:nn { Unknown~key~for~marker } { Unknown-key. \par The-key-'marker / \l_keys_key_str'-is-unknown.\par The-available-keys-of-the-family-'marker'-are-(in- alphabetic-order):-beginning,-end-and-include-lines.\par That-key-will-be-ignored. }
\@@_msg_new:nn { bad-range-specification } { Incompatible-keys.\par You-can't-specify-the-range-of-lines-to-include-by-using-both- markers-and-explicit-number-of-lines.\par Your-whole-file-'\l_@@_file_name_str'-will-be-included. }
Your code is not syntactically correct. It won't be printed in the PDF file.

The range '\l_@@_begin_range_str'-provided-to-the-command\token_to_str:N \PitonInputFile\ has not been found.

The whole file '\l_@@_file_name_str'-will be inserted.

Marker-not-found.\ The-marker-of-end-of-the-range-'\l_@@_end_range_str'-provided-to-the-command\token_to_str:N \PitonInputFile\ has not been found. The file '\l_@@_file_name_str'-will be inserted till the end.

Unknown-file. \The-file#'\l_@@_file_name_str'-is unknown. Your command\token_to_str:N \PitonInputFile\ will be discarded.

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):
  auto-gobble,-
  background-color,-
  break-lines,-
  break-lines-in-piton,-
  break-lines-in-Piton,-
  continuation-symbol,-
  continuation-symbol-on-indentation,-
  end-of-broken-line,-
  end-range,-
  env-gobble,-
  gobble,-
  identifiers,-
  indent-broken-lines,-
  language,-
  left-margin,-
  line-numbers/-,
  marker/-,
  prompt-background-color,-
  resume,-
show-spaces,-
show-spaces-in-strings,-
splittable,-
tabs-auto-gobble,-
tab-size-and-width.
}
\@@_msg_new:nn { label-with-lines-numbers }
{ 
You-can't-use-the-command-\token_to_str:N \label
because-the-key-'line-numbers'-is-not-active.\\nIf-you-go-on,-that-command-will-ignored.
}
\@@_msg_new:nn { cr-not-allowed }
{ 
You-can't-put-any-carriage-return-in-the-argument-
of-a-command-\c_backslash_str
\l_@@_beamer_command_str\ within-an-
environment-of-'piton'.-You-should-consider-using-the-
corresponding-environment.\\nThat-error-is-fatal.
}
\@@_msg_new:nn { overlay-without-beamer }
{ 
You-can't-use-an-argument-<...>-for-your-command-
\token_to_str:N \PitonInputFile\ because-you-are-not-
in-Beamer.\\nIf-you-go-on,-that-argument-will-be-ignored.
}
\@@_msg_new:nn { Python-error }
{ A-Python-error-has-been-detected. }

8.2.13 We load piton.lua

\hook_gput_code:nnn { begindocument } { . }
{ \lua_now:e { require("piton.lua") } }
{/STY}

8.3 The Lua part of the implementation

The Lua code will be loaded via a {luacode*} environment. The 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.

{LUAd}
if piton.comment_latex == nil then piton.comment_latex = "\>" end
piton.comment_latex = "#" .. piton.comment_latex

The following functions are an easy way to safely insert braces ({ and }) in the TeX flow.

function piton.open_brace ()
tex.sprint("{")
end
function piton.close_brace ()
tex.sprint("}")
end
8.3.1 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.

```lua
local Cf, Cs, Cg, Cmt, Cb = lpeg.Cf, lpeg.Cs, lpeg.Cg, lpeg.Cmt, lpeg.Cb
local R = lpeg.R
```

The function `Q` takes in as argument a pattern and returns a `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 `piton` will typeset verbatim (thanks to the catcode “other”).

```lua
local function Q(pattern)
    return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
end
```

The function `L` takes in as argument a pattern and returns a `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 “LaTeX comments” in the environments `{Piton}` and the elements between `begin-escape` and `end-escape`. That function won’t be much used.

```lua
local function L(pattern)
    return Ct ( C ( pattern ) )
end
```

The function `Lc` (the `c` is for `constant`) takes in as argument a string and returns a `lpeg` with does a 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 `piton`). That function will be widely used.

```lua
local function Lc(string)
    return Cc ( { luatexbase.catcodetables.expl , string } )
end
```

The function `K` creates a `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 Lua string corresponding to the name of a `piton` style and the second element is a pattern (that is to say a `lpeg` without capture).

```lua
local function K(style, pattern)
    return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
end
```

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

```lua
local function WithStyle(style,pattern)
    return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
end
```

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

```lua
local function WithStyle(style,pattern)
    return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
end
```
The following LPEG catches the Python chunks which are in LaTeX escapes (and that chunks will be considered as normal LaTeX constructions). Since the elements that will be catched must be sent to LaTeX with standard LaTeX catcodes, we put the capture (done by the function \texttt{C}) in a table (by using \texttt{Ct}, which is an alias for \texttt{lpeg.Ct}) without number of catcode table at the first component of the table.

```lua
Escape = P ( false )
if piton.begin_escape == nil
then
Escape = P(piton.begin_escape)
* L ( ( 1 - P(piton.end_escape) ) ^ 1 )
* P(piton.end_escape)
end

EscapeMath = P ( false )
if piton.begin_escape_math == nil
then
EscapeMath = P(piton.begin_escape_math)
* Lc ( "\ensuremath{" )
* L ( ( 1 - P(piton.end_escape_math) ) ^ 1 )
* Lc ( ")" )
* P(piton.end_escape_math)
end
```

The following line is mandatory.

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

The basic syntactic LPEG

```lua
local alpha, digit = lpeg.alpha, lpeg.digit
local space = P " "
local alphanum = letter + digit
```

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

```lua
local identifier = letter * alphanum ^ 0
```

On the other hand, the LPEG \texttt{Identifier} (with a capital) also returns a capture.

```lua
local Identifier = K ( 'Identifier' , identifier)
```

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

Here is the first use of our function \texttt{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 \texttt{Number}. The name of the style is provided as a Lua string in the second argument of the function \texttt{K}. By convention, we use single quotes for delimiting the Lua strings which are names of piton styles (but this is only a convention).

```lua
local Number =
K ( 'Number' ,
( digit"1" * P "." * digit"0" + digit"0" * P "." * digit"1" + digit"1" )
)
We recall that `piton.begin_escape` and `piton_end_escape` are Lua strings corresponding to the keys `begin-escape` and `end-escape`.

```lua
local Word
if piton.begin_escape ~= nil -- before : ''
then Word = Q ( ( 1 - space - P(piton.begin_escape) - P(piton.end_escape) )
    - S "\"r()" - digit ) ^ 1 )
else Word = Q ( ( 1 - space ) - S "\"r()" - digit ) ^ 1 )
end

local Space = ( Q " " ) ^ 1
local SkipSpace = ( Q " " ) ^ 0
local Punct = Q ( S ".;:;!" )
local Tab = P \t" * Lc ( '\l_@@_tab_tl' )
local SpaceIndentation = Lc ( '\\@@_an_indentation_space:' ) * ( Q " " )
local Delim = Q ( S ":[(]" )

The following `lpeg` catches a space (U+0020) and replace it by `\l_@@_space_tl`. It will be used in the strings. Usually, `\l_@@_space_tl` will contain a space and therefore there won’t be difference. However, when the key `show-spaces-in-strings` is in force, `\l_@@_space_tl` contains `\r` (U+2423) in order to visualize the spaces.

```lua
local VisualSpace = space * Lc \"l_@@_space_tl"
```

If the classe Beamer is used, some environemnts and commands of Beamer are automatically detected in the listings of `piton`.

```lua
local Beamer = P ( false )
local BeamerBeginEnvironments = P ( true )
local BeamerEndEnvironments = P ( true )
if piton_beamer
then
% \bigskip
% The following function will return a \textsc{lpeg} which will catch an
% environment of Beamer (supported by \pkg{piton}), that is to say \{uncover\},
% \{only\}, etc.
% \begin{macrocode}
local BeamerNamesEnvironments =
    P "uncoverenv" + P "onlyenv" + P "visibleenv" + P "invisibleenv"
    + P "alertenv" + P "actionenv"
BeamerBeginEnvironments =
    ( space ^ 0 *
    L
    ( P "\begin{" * BeamerNamesEnvironments * "})"
    * ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1
    )
    * P "\r"
    ) ^ 0
BeamerEndEnvironments =
    ( space ^ 0 *
    L ( P "\end{" * BeamerNamesEnvironments * P "}")
    * P "\r"
    ) ^ 0

55
The following function will return a LPEG which will catch an environment of Beamer (supported by \texttt{piton}), that is to say \texttt{\textbackslash begin\{uncoverenv\}, etc. The argument \texttt{lpeg} should be \texttt{MainLoopPython}, \texttt{MainLoopC}, etc.

```plaintext
function OneBeamerEnvironment(name, lpeg)
    return
    Ct ( Cc "Open"
        * C ( 
            P ( \textbackslash begin\{ .. name .. \} \textbackslash\}
            + ( P "<" + ( 1 - P => ) ^ 0 + P ">" ) ^ -1 
        )
        * Cc ( \textbackslash end\{ .. name .. \}\textbackslash\} 
    )
    )
    * ( 
        C ( ( 1 - P ( \textbackslash end\{ .. name .. \} \textbackslash\}) ) ^ 0 )
        / ( function (s) return lpeg : match(s) end )
    )
    * P ( \textbackslash end\{ .. name .. \}\textbackslash\} ) * Ct ( Cc "Close" )
end
end

local languages = { }
```

8.3.2 The LPEG python

Some strings of length 2 are explicit because we want the corresponding ligatures available in some fonts such as \textit{Fira Code} to be active.

```plaintext
local Operator =
    K ( 'Operator' ,
        P "\!=" + P "><" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P ":="
            + P "//" + P "**" + S "-~+/*%=<>&.@|"
    )

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

local Keyword =
    K ( 'Keyword' ,
        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" )
            + K ( 'Keyword.Constant' ,P "True" + P "False" + P "None" )

local Builtin =
    K ( 'Name.Builtin' ,
        P ".import_" + P "abs" + P "all" + P "any" + P "bin" + P "bool"
            + P "bytearray" + P "bytes" + P "chr" + P "classmethod" + P "compile"
            + P "complex" + P "delattr" + P "dict" + P "dir" + P "divmod"
            + P "enumerate" + P "eval" + P "filter" + P "float" + P "format"
            + P "frozenset" + P "getattr" + P "globals" + P "hasattr" + P "hash"
            + P "hex" + P "id" + P "input" + P "int" + P "isinstance" + P "issubclass"
            + P "iter" + P "len" + P "list" + P "locals" + P "map" + P "max"
            + P "memorize" + P "min" + P "next" + P "object" + P "oct" + P "open"
            + P "ord" + P "pow" + P "print" + P "property" + P "range" + P "repr"
            + P "reversed" + P "round" + P "set" + P "slice" + P "sorted"
            + P "staticmethod" + P "str" + P "sum" + P "super" + P "tuple" + P "type"
            + P "vars" + P "zip" )
```

56
local Exception = 
    K ( 'Exception' , 
        P "ArithmeticError" + P "AssertionError" + P "AttributeError" 
        + P "BaseException" + P "BufferError" + P "BytesWarning" + P "DeprecationWarning" 
        + P "EOFError" + P "EnvironmentError" + P "Exception" + P "FloatingPointError" 
        + P "FutureWarning" + P "GeneratorExit" + P "IOError" + P "ImportError" 
        + P "ImportWarning" + P "IndentationError" + P "IndexError" + P "KeyError" 
        + P "KeyboardInterrupt" + P "LookupError" + P "MemoryError" + P "NameError" 
        + P "NotImplementedError" + P "OSError" + P "OverflowError" 
        + P "PendingDeprecationWarning" + P "ReferenceError" + P "ResourceWarning" 
        + P "RuntimeError" + P "RuntimeWarning" + P "StopIteration" 
        + P "SyntaxError" + P "SyntaxWarning" + P "SystemError" + P "SystemExit" 
        + P "TabError" + P "TypeError" + P "UnboundLocalError" + P "UnicodeDecodeError" 
        + P "UnicodeEncodeError" + P "UnicodeError" + P "UnicodeTranslateError" 
        + P "UnicodeWarning" + P "UserWarning" + P "ValueError" + P "Warning" 
        + P "Warning" + P "WindowsError" + P "ZeroDivisionError" 
        + P "BlockingIOError" + P "ChildProcessError" + P "ConnectionError" 
        + P "BrokenPipeError" + P "ConnectionAbortedError" + P "ConnectionRefusedError" 
        + P "ConnectionResetError" + P "FileExistsError" + P "FileNotFoundError" 
        + P "InterruptedError" + P "IsADirectoryError" + P "NotADirectoryError" 
        + P "PermissionError" + P "ProcessLookupError" + P "TimeoutError" 
        + P "StopAsyncIteration" + P "ModuleNotFoundError" + P "RecursionError" ) 

local RaiseException = K ( 'Keyword' , P "raise" ) * SkipSpace * Exception * Q ( P "(" )

In Python, a “decorator” is a statement whose begins by @ which patches the function defined in the following statement.

local Decorator = K ( 'Name.Decorator' , P "@" * letter^1 )

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 ( 'Keyword' , P "class" ) * Space * K ( 'Name.Class' , identifier )

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

    local ImportAs = 
        K ( 'Keyword' , P "import" ) 
        * Space 
        * K ( 'Name.Namespace' , 
            identifier * ( P "." * identifier ) ^ 0 ) 
        * ( 
            ( Space * K ( 'Keyword' , P "as" ) * Space 
                * K ( 'Name.Namespace' , identifier ) )
            +
            ( SkipSpace * Q ( P "," ) * SkipSpace 
                * K ( 'Name.Namespace' , identifier ) ) ^ 0 
        )

57
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 caught by the LPEG ImportAs.

Example: from math import pi

```plaintext
local FromImport =
  local FromImport =
    K ( 'Keyword' , P "from" )
    * Space * K ( 'Name.Namespace' , identifier )
    * Space * K ( 'Keyword' , P "import" )
```

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>&quot;text&quot;</td>
</tr>
<tr>
<td>Long</td>
<td>&quot;&quot;&quot;text&quot;&quot;&quot;&quot;</td>
</tr>
</tbody>
</table>

We have also to deal with the interpolations in the f-strings. Here is an example of a f-string with an interpolation and a format instruction\(^{22}\) in that interpolation:

```plaintext
f'Total price: {total+1:.2f} €'
```

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

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

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.\(^{24}\)

```plaintext
local SingleShortString =
  WithStyle ( 'String.Short' ,
  WithStyle ( 'String.Short' ,
    Q ( P "f" + P "F" )
    Q ( P "f" + P "F" )
    * ( K ( 'String.Interpol' , P "{" ) * K ( 'Interpol.Inside' , ( 1 - S ":" ) ^ 0 ) * Q ( P ":" * ( 1 - S ":" ) ^ 0 ) ^ -1
    * K ( 'String.Interpol' , P "}" )
    +
    VisualSpace
    +
```

\(^{22}\)There is no special piton style for the formatting instruction (after the colon): the style which will be applied will be the style of the encompassing string, that is to say String.Short or String.Long.

\(^{24}\)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.
Now, we deal with the standard strings of Python, but also the “raw strings”.

\[
\begin{align*}
  Q& (\ P"\" + P "r" + P "R") \\
& * ( Q ( ( P "\" + 1 - S " \"r\" ) ^ 1 ) \\
& + VisualSpace \\
& + PercentInterpol \\
& + Q ( P "%" ) \\
& ) ^ 0 \\
& * Q ( P "\" )
\end{align*}
\]

local DoubleShortString =
  WithStyle ( 'String.Short' ,
    Q ( P "f\" + P "F\" )
* ( ( K ( 'String.Interpol' , P "{" ) \\
  * Q ( ( 1 - S "}:" ) ^ 0 , 'Interpol.Inside' ) \\
  * ( K ( 'String.Interpol' , P ":" ) * Q ( (1 - S ":}:" ) ^ 0 ) ) ^ -1 \\
  * K ( 'String.Interpol' , P "}" ) \\
  + VisualSpace \\
  + Q ( ( P "\" + P "\" + P "R\" ) \\
  * ( Q ( ( P "\" + 1 - S " "\"r\" ) ^ 1 ) \\
  + VisualSpace \\
  + PercentInterpol \\
  + Q ( P "%" ) \\
  ) ^ 0 \\
  * Q ( P "\" )
  ) \\
local ShortString = SingleShortString + DoubleShortString

\textbf{Beamer} \quad The following pattern \texttt{balanced_braces} will be used for the (mandatory) argument of the commands \texttt{\only} and \textit{al.} of Beamer. 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 \textit{stricto sensu} only).

local balanced_braces =
P { "E" ,
  E =
  ( 
  P "{" * V "E" * P "}" \\
  + ShortString \\
  + ( 1 - S "}" ) \\
  ) ^ 0 
}

if piton_beamer
then
  Beamer =
  L ( P "\pause" * ( P "[" * ( 1 - P "]") ^ 0 * P "]") ^ -1 )
+
For \texttt{\textbackslash{}alt}, the specification of the overlays (between angular brackets) is mandatory.

\begin{verbatim}
L ( P "\textbackslash{}alt" )
  * P "<" * (1 - P ">") ^ 0 * P ">"
  * P "{"  
  * K ( 'ParseAgain.noCR', balanced_braces )  
  * L ( P "}" )
  * K ( 'ParseAgain.noCR', balanced_braces )  
  * L ( P "}" )
+ L ( P "}" )
\end{verbatim}

For \texttt{\textbackslash{}temporal}, the specification of the overlays (between angular brackets) is mandatory.

\begin{verbatim}
L ( P "\textbackslash{}temporal" )
  * P "<" * (1 - P ">") ^ 0 * P ">"
  * P "{"  
  * K ( 'ParseAgain.noCR', balanced_braces )  
  * L ( P "}" )
  * K ( 'ParseAgain.noCR', balanced_braces )  
  * L ( P "}" )
  * K ( 'ParseAgain.noCR', balanced_braces )  
  * L ( P "}" )
end
\end{verbatim}

\textbf{EOL} The following LPEG will detect the Python prompts when the user is typesetting an interactive session of Python (directly or through \texttt{\textbackslash{}pyconsole}} of pylatex). We have to detect that prompt twice. The first detection (called \textit{hasty detection}) will be before the \texttt{\textbackslash{}begin\_line}} because you want to trigger a special background color for that row (and, after the \texttt{\textbackslash{}begin\_line}}; it’s too late to change de background).

\begin{verbatim}
local PromptHastyDetection = ( # ( P ">>>" + P "{..." ) * Lc ( '\@@\_prompt:!' ) ) ^ -1
\end{verbatim}

We remind that the marker \# of LPEG specifies that the pattern will be detected but won’t consume any character.

With the following LPEG, a style will actually be applied to the prompt (for instance, it’s possible to decide to discard these prompts).

\begin{verbatim}
local Prompt = K ( 'Prompt' , ( ( P ">>>" + P "{..." ) * P " " ^ -1 ) ^ -1 )
\end{verbatim}
The following `lpeg` `EOL` is for the end of lines.

1581 local EOL =
1582 P "\r"
1583 *
1584 ( space^0 * -1 )
1585 +

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:`.

1587 Ct (  
1588    Cc "EOL"
1589    *
1590    Ct (  
1591        Lc "\@@_end_line:"
1592        * BeamerEndEnvironments
1593        * BeamerBeginEnvironments
1594        * PromptHastyDetection
1595        * Lc "\@@_newline: \@@_begin_line:"
1596        * Prompt
1597    )
1598 )
1599 *
1600 SpaceIndentation ^ 0

The long strings

1602 local SingleLongString =
1603 WithStyle ( 'String.Long' ,  
1604    ( Q ( S "fF" * P "'" )  
1605    * (  
1606      K ( 'String.Interpol' , P "("  
1607      * K ( 'Interpol.Inside' , ( 1 - S "\r" - P "'" ) ^ 0 )  
1608      * Q ( P ";" * (1 - S ":\r" - P "'" ) ^ 0 ) ^ -1  
1609      * K ( 'String.Interpol' , P ")" )  
1610    +  
1611    Q ( ( 1 - P "'" ) - S "{}\r" ) ^ 1  
1612    +  
1613    EOL  
1614 ) ^ 0  
1615    +  
1616    Q ( ( S "rR" ) ^ -1 * P "'" )  
1617    * (  
1618    Q ( ( 1 - P "'" ) - S "rR" ) ^ 1  
1619    +  
1620    PercentInterpol  
1621    +  
1622    P "%"  
1623    +  
1624    EOL  
1625 ) ^ 0  
1626 )  
1627 * Q ( P "'" )
1628

1630 local DoubleLongString =
1631 WithStyle ( 'String.Long' ,
1632    (  

25Remember 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 a LPEG for the Python docstrings. That LPEG will be used in the LPEG DefFunction which deals with the whole preamble of a function definition (which begins with def).

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

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).
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).

```plaintext
local expression =
P { "E",
    E = ( P "^" * ( P "\\" + 1 - S "\\r" ) ^ 0 * P "^"
            + P "(" * P "\" + 1 - S "\"r" ) ^ 0 * P "\"
            + P "(" * P "F" * P ")"
            + P "[" * P "F" * P "]"
            + ( 1 - S "{()\r" ) ) ^ 0 ,
    F = ( P "^" * P "F" * P ")"
            + P "(" * P "F" * P ")"
            + P "[" * P "F" * P "]"
            + ( 1 - S "{()\r" ) ) ^ 0

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

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 ( 'InitialValues' , P "=" * expression )
     + Q ( P ":" ) * SkipSpace * K ( 'Name.Type' , letter ^ 1 )
   ) ^ -1

local Params = ( Param * ( Q "," * 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 ( 'Keyword' , P "def" )
   * Space
   * K ( 'Name.Function.Internal' , identifier )
   * SkipSpace
   * Q ( P ":" ) * Params * Q ( P ")" )
   * SkipSpace
   * ( Q ( P "->" ) * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1

Here, we need a piton style ParseAgain 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 ( 'ParseAgain' , ( 1 - S ":\r" )0 )
   * Q ( 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, \texttt{CommentLaTeX} must appear before \texttt{Comment}: there is no commutativity of the addition for the \textit{parsing expression grammars} (PEG).

If the word \texttt{def} is not followed by an identifier and parenthesis, it will be caught as keyword by the LPEG \texttt{Keyword} (useful if, for example, the final user wants to speak of the keyword \texttt{def}).

\textbf{Miscellaneous}

\begin{verbatim}
local ExceptionInConsole = Exception * Q (( 1 - P "\r" ) ^ 0 ) * EOL
\end{verbatim}

\textbf{The main LPEG for the language Python}  
First, the main loop :

\begin{verbatim}
local MainPython =
  EOL
  + Space
  + Tab
  + Escape + EscapeMath
  + CommentLaTeX
  + Beamer
  + LongString
  + Comment
  + ExceptionInConsole
  + Delim
  + Operator
  + OperatorWord * ( Space + Punct + Delim + EOL + -1 )
  + ShortString
  + Punct
  + FromImport
  + RaiseException
  + DefFunction
  + DefClass
  + Keyword * ( Space + Punct + Delim + EOL + -1 )
  + Decorator
  + BuiltIn * ( Space + Punct + Delim + EOL + -1 )
  + Identifier
  + Number
  + Word

Ici, il ne faut pas mettre local !

\begin{verbatim}
local MainLoopPython =
  ( ( space^1 * -1 )
  + MainPython
  ) ^ 0
\end{verbatim}

We recall that each line in the Python code to parse will be sent back to LaTeX between a pair \texttt{\@@_begin_line: – \@@_end_line:}.\footnote{Remember that the \texttt{\@@_end_line:} must be explicit because it will be used as marker in order to delimit the argument of the command \texttt{\@@_begin_line:}}

\begin{verbatim}
local python = P ( true )
\end{verbatim}

\begin{verbatim}
python =
  Ct ( ( space - P "\r" ) "0 + P "\r" ) ^ -1
  * BeamerBeginEnvironments
  * PromptHastyDetection
  * Lc '\@@_begin_line: '
  * Prompt
  * SpaceIndentation ^ 0
  * MainLoopPython
  * -1
  * Lc '\@@_end_line: '
)  
\end{verbatim}
The identifiers caught by `cap_identifier` begin with a cap. In OCaml, it’s used for the constructors of types and for the modules.

The identifiers which begin with a lower case letter or an underscore are used elsewhere in OCaml.

Now, we deal with the records because we want to catch the names of the fields of those records in all circumstances.

Now, we deal with the notations with points (eg: `List.length`). In OCaml, such notation is used for the fields of the records and for the modules.
The following exceptions are exceptions in the standard library of OCaml (Stdlib).

```ocaml
local Exception =
K ('Exception',
P "Division_by_zero" + P "End_of_File" + P "Failure"
+ P "Invalid_argument" + P "Lazy_limit" + P "Not_found"
+ P "Out_of_memory" + P "Stack_overflow" + P "Sys_blocked_io"
+ P "Sys_error" + P "Undefined_recursive_module")
```

The characters in OCaml

```ocaml
local Char =
K ('String.Short', P "" * ( 1 - P "" ) 0 + P "\"" ) * P "" )
```

Beamer

```ocaml
local balanced_braces =
P { "E",
E =
{
P "{" * V "E" * P "}"
+ P "\"" * ( 1 - S "\"" ) 0 * P "\"" -- OCaml strings
+ ( 1 - S "{\}" )
```
if piton_beamer
then
  Beamer =
  L ( P "\pause" * ( P "[" * ( 1 - P "]" ) ^ 0 * P "]" ) ^ -1 )
  +
  Ct ( Cc "Open"
    * C {
      ( P "\uncover" + P "\only" + P "\alert" + P "\visible"
        + P "\invisible" + P "\action"
      )
      * ( P "<" * (1 - P ">") ^ 0 * P ">" ) ^ -1
      * P "("
    }
    * Cc ")")
    * ( C ( balanced_braces ) / (function (s) return MainLoopOCaml:match(s) end ) )
    * P ")" * Ct ( Cc "Close" )
  + OneBeamerEnvironment ( "uncoverenv" , MainLoopOCaml )
  + OneBeamerEnvironment ( "onlyenv" , MainLoopOCaml )
  + OneBeamerEnvironment ( "visibleenv" , MainLoopOCaml )
  + OneBeamerEnvironment ( "invisibleenv" , MainLoopOCaml )
  + OneBeamerEnvironment ( "alertenv" , MainLoopOCaml )
  + OneBeamerEnvironment ( "actionenv" , MainLoopOCaml )
  +
  L ( P "\alt"
    * P "<" * (1 - P ">") ^ 0 * P ">"
    * P "("
    )
    * K ( 'ParseAgain.noCR' , balanced_braces )
    * L ( P ")" )
    * K ( 'ParseAgain.noCR' , balanced_braces )
    * L ( P "(" )
  )
  L ( P "\temporal"
    * P "<" * (1 - P ">") ^ 0 * P ">"
    * P "("
    )
    * K ( 'ParseAgain.noCR' , balanced_braces )
    * L ( P ")" )
    * K ( 'ParseAgain.noCR' , balanced_braces )
    * L ( P ")" )
    * K ( 'ParseAgain.noCR' , balanced_braces )
    * L ( P "(" )
  )
end

EOL
local EOL =
P "\r"
* ( space*0 * -1 )
+
The strings en OCaml We need a pattern `ocaml_string` without captures because it will be used within the comments of OCaml.

```ocaml
local ocaml_string = Q ( P "\"" ) * ( VisualSpace + Q ( ( 1 - S " \r" ) ^ 1 ) + EOL ) ^ 0 * Q ( P "\"" ) local String = WithStyle ( 'String.Long', ocaml_string )
```

Now, the “quoted strings” of OCaml (for example `{ext|Essai|ext}`). For those strings, we will do two consecutive analysis. First an analysis to determine the whole string and, then, an analysis for the potential visual spaces and the EOL in the string.

The first analysis require a match-time capture. For explanations about that programation, see the paragraphe Lua’s long strings in www.inf.puc-rio.br/~roberto/lpeg.

```ocaml
local ext = ( R "az" + P "." ) ^ 0 local open = "{: Cg(ext, 'init') * "|
local close = "|" * C(ext) * "}
local closeeq = Cmt ( close * Cb('init'), function (s, i, a, b) return a==b end )
```

The LPEG `QuotedStringBis` will do the second analysis.

```ocaml
local QuotedStringBis = WithStyle ( 'String.Long' , ( VisualSpace + Q ( ( 1 - S " \r" ) ^ 1 ) + EOL ) ^ 0 )
```

We use a “function capture” (as called in the official documentation of the LPEG) in order to do the second analysis on the result of the first one.

```ocaml
local QuotedString = C ( open * ( 1 - closeeq ) ^ 0 * close ) / ( function (s) return QuotedStringBis : match(s) end )
```
The comments in the OCaml listings  In OCaml, the delimiters for the comments are (* and *). There are unsymmetrical and OCaml allow those comments to be nested. That’s why we need a grammar.
In these comments, we embed the math comments (between $ and $) and we embed also a treatment for the end of lines (since the comments may be multi-lines).

```ocaml
local Comment = WithStyle ( 'Comment' ,
  P {
    "A" ,
    A = Q "(*" *
        ( V "A"
          + Q ( ( 1 - P "(*" - P ")" - S "\r$" ) ^ 1 ) -- $ 
          + ocaml_string
          + P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$" -- $ 
          + EOL
          ) ^ 0
        + Q "*)"
    )
  }
)

The DefFunction

```ocaml
local balanced_parens = P { "E" ,
  E =
  ( P "(" *
      ( V "E" *
        P ")"
        +
        ( 1 - S "()" )
      ) ^ 0
  )
)
local Argument =
K ( 'Identifier' , identifier )
+ Q "(" * SkipSpace
  * K ( 'Identifier' , identifier ) * SkipSpace
+ Q ":" * SkipSpace
  * K ( 'Name.Type' , balanced_parens ) * SkipSpace
+ Q ")"

Despite its name, then lpeg DefFunction deals also with let open which opens locally a module.

```ocaml
local DefFunction =
K ( 'Keyword' , P "let open" )
* Space
  * K ( 'Name.Module' , cap_identifier )
+ K ( 'Keyword' , P "let rec" + P "let" + P "and" )
  * Space
  * K ( 'Name.Function.Internal' , identifier )
  * Space
  *( Q ":=" * SkipSpace * K ( 'Keyword' , P "function" )
    +
    Argument
  )
  * ( SkipSpace * Argument ) ^ 0
  *
  *( SkipSpace
    + Q ":="
    * K ( 'Name.Type' , ( 1 - P ":=" ) ^ 0
    ) ^ -1
  )
```
The **DefModule**  The following LPEG will be used in the definitions of modules but also in the definitions of *types* of modules.

```
local DefModule =
  K ( 'Keyword', P "module" ) * Space
  *
  (  
    K ( 'Keyword', P "type" ) * Space
    * K ( 'Name.Type', cap_identifier )
    +
    K ( 'Name.Module', cap_identifier ) * SkipSpace
    *
    ( 
      Q "(" * SkipSpace
      * K ( 'Name.Module', cap_identifier ) * SkipSpace
      * Q ":" * SkipSpace
      * K ( 'Name.Type', cap_identifier ) * SkipSpace
    ) ^ 0
    * Q ")"
  ) ^ -1
*
( 
  Q "=" * SkipSpace
  * K ( 'Name.Module', cap_identifier ) * SkipSpace
  * Q "("  
  * K ( 'Name.Module', cap_identifier ) * SkipSpace
    *
    ( 
      Q ","  
      * K ( 'Name.Module', cap_identifier ) * SkipSpace
    ) ^ 0
    * Q ")"
  ) ^ -1
)  
+  
K ( 'Keyword', P "include" + P "open" )
* Space + K ( 'Name.Module', cap_identifier )
```

The parameters of the types

```
local TypeParameter = K ( 'TypeParameter', P "::" * alpha * # ( 1 - P ":" ) )
```

The main LPEG for the language **OCaml**  First, the main loop :

```
MainOCaml =  
  EOL
  + Space
  + Tab
  + Escape + EscapeMath
  + Beamer
  + TypeParameter
  + String + QuotedString + Char
  + Comment
  + Delim
  + Operator
```

70
We recall that each line in the Python code to parse will be sent back to LaTeX between a pair \@@_begin_line: – \@@_end_line:.

languages['ocaml'] = ocaml

8.3.4 The LPEG language C

Some strings of length 2 are explicit because we want the corresponding ligatures available in some fonts such as Fira Code to be active.

local Operator = K ('Operator',
  P '!=' + P '==' + P '<<' + P '>>' + P '<=' + P '>='
    + P '||' + P '&&' + S '-~+/*% /=<>&.@|!'
  )
)

local Keyword =
K ('Keyword',
  P 'alignas' + P 'asm' + P 'auto' + P 'break' + P 'case' + P 'catch'
  + P 'class' + P 'const' + P 'constexpr' + P 'continue'
  + P 'decltype' + P 'do' + P 'else' + P 'enum' + P 'extern'
  + P 'for' + P 'goto' + P 'if' + P 'nexcept' + P 'private' + P 'public'
  + P 'register' + P 'restricted' + P 'return' + P 'static'
  + P 'static_assert'
  + P 'strict' + P 'switch' + P 'thread_local' + P 'throw' + P 'try'

\(\textit{Remember that the \@@_end_line: must be explicit because it will be used as marker in order to delimit the argument of the command \@@_begin_line:}\)
+ P "typedef" + P "union" + P "using" + P "virtual" + P "volatile"
+ P "while"

+ K ( 'Keyword.Constant' ,
P "default" + P "false" + P "NULL" + P "nullptr" + P "true"
)

local Builtin =
K ( 'Name.Builtin' ,
P "alignof" + P "malloc" + P "printf" + P "scanf" + P "sizeof"
)

local Type =
K ( 'Name.Type' ,
P "bool" + P "char" + P "char16_t" + P "char32_t" + P "double"
+ P "float" + P "int" + P "int8_t" + P "int16_t" + P "int32_t"
+ P "int64_t" + P "long" + P "short" + P "signed" + P "unsigned"
+ P "void" + P "wchar_t"
)

local DefFunction =
Type
* Space
* K ( 'Name.Function.Internal' , identifier )
* SkipSpace
* # P "("

We remind that the marker # of LPEG specifies that the pattern will be detected but won't consume any character.

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 ( 'Keyword' , P "class" ) * Space * K ( 'Name.Class' , identifier )

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 strings of C
local String =
WithStyle ( 'String.Long' ,
Q "\""
* ( VisualSpace
  + K ( 'String.Interpol' ,
     P "\%" * ( S "difcspxXou" + P "ld" + P "li" + P "hd" + P "hi" )
    )
  + Q ( ( P "\\\"" + 1 - S " \"" ) ^ 1 )
    ) ^ 0
  ) * Q "\""
)

Beamer The following LPEG balanced_braces will be used for the (mandatory) argument of the commands \only and al. of Beamer. 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).

local balanced_braces =
P { "E" ,
E =
(
if piton_beamer
    then
    Beamer =
        L ( P "\pause" * ( P "[" * ( 1 - P "]" ) ^ 0 * P "]" ) ^ -1 )
        +
        Ct ( Cc "Open"
            * C {
                P "\uncover" + P "\only" + P "\alert" + P "\visible"
                + P "\invisible" + P "\action"
                )
                * ( P "<" * (1 - P ">" ) ^ 0 * P ">" ) ^ -1
                * P "{" 
            } 
            * Cc "})"
        ) * ( C ( balanced_braces ) / (function (s) return MainLoopC:match(s) end ) )
        * P ")" * Ct ( Cc "Close" )
        + OneBeamerEnvironment ( "uncoverenv" , MainLoopC )
        + OneBeamerEnvironment ( "onlyenv" , MainLoopC )
        + OneBeamerEnvironment ( "visibleenv" , MainLoopC )
        + OneBeamerEnvironment ( "invisibleenv" , MainLoopC )
        + OneBeamerEnvironment ( "alertenv" , MainLoopC )
        + OneBeamerEnvironment ( "actionenv" , MainLoopC )
        +
        L ( 

    For \alt, the specification of the overlays (between angular brackets) is mandatory.

        ( P "\alt" )
        * P "<" * (1 - P ">" ) ^ 0 * P ">" 
        * P "{" 
        )
        * K ( 'ParseAgain.noCR' , balanced_braces )
        * L ( P "}" )
        * K ( 'ParseAgain.noCR' , balanced_braces )
        * L ( P "}" )
        +
        L ( 

    For \temporal, the specification of the overlays (between angular brackets) is mandatory.

        ( P "\temporal" )
        * P "<" * (1 - P ">" ) ^ 0 * P ">" 
        * P "{" 
        )
        * K ( 'ParseAgain.noCR' , balanced_braces )
        * L ( P "}" )
        * K ( 'ParseAgain.noCR' , balanced_braces )
        * L ( P "}" )
        * K ( 'ParseAgain.noCR' , balanced_braces )
        * L ( P "}" )
end
The following LPEG will detect the Python prompts when the user is typesetting an interactive session of Python (directly or through `{pyconsole}` of pylatex). We have to detect that prompt twice. The first detection (called *hasty detection*) will be before the `{\@@_begin_line}` because you want to trigger a special background color for that row (and, after the `{\@@_begin_line}`, it’s too late to change the background).

```plaintext
local PromptHastyDetection = ( # ( P ">>>" + P "..." ) * Lc ( '\@@_prompt:' ) ) ^ -1
```

We remind that the marker # of LPEG specifies that the pattern will be detected but won’t consume any character.

With the following LPEG, a style will actually be applied to the prompt (for instance, it’s possible to decide to discard these prompts).

```plaintext
local Prompt = K ( 'Prompt', ( ( P ">>>" + P "...") * P " " ^ -1 ) ^ -1 )
```

The following LPEG EOL is for the end of lines.

```plaintext
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:}`.

```plaintext
Ct ( Cc "EOL" * Ct ( Lc "\@@_end_line:" * BeamerEndEnvironments * BeamerBeginEnvironments * PromptHastyDetection * Lc "\@@_newline: \@@_begin_line:" * Prompt * ) ) )
```

The directives of the preprocessor

```plaintext
local Preproc = K ( 'Preproc', P "#" * (1 - P "\r") ^ 0 ) * ( EOL + -1 )
```

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

```plaintext
local CommentMath = P "$" * K ( 'Comment.Math', ( 1 - S "$\r" ) ^ 1 ) * P "$"
```

```plaintext
local Comment = WithStyle ( 'Comment', Q ( P "//" ) * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) *
```

28Remember 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:}`.
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).

The main LPEG for the language C  First, the main loop :

We recall that each line in the C code to parse will be sent back to LaTeX between a pair \@@_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:
function piton.Parse(language, code)
    local t = languages[language]:match(code)
    if t == nil
        tex.sprint("\PitonSyntaxError")
        return -- to exit in force the function
    end
    local left_stack = {}
    local right_stack = {}
    for _, one_item in ipairs(t) do
        if one_item[1] == "EOL"
            for _, s in ipairs(right_stack) do tex.sprint(s) end
            for _, s in ipairs(one_item[2]) do tex.tprint(s) end
            for _, s in ipairs(left_stack) do tex.sprint(s) end
        else
            if one_item[1] == "Open"
                for _, s in ipairs(right_stack) do tex.sprint(s) end
                for _, s in ipairs(one_item[2]) do tex.tprint(s) end
                for _, s in ipairs(left_stack) do tex.sprint(s) end
            else
                if one_item[1] == "Close"
                    tex.sprint( right_stack[#right_stack] )
                    left_stack[#left_stack] = nil
                    right_stack[#right_stack] = nil
                else
                    tex.tprint(one_item)
                end
            end
        end
    end
end

8.3.5 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 corresponding to the considered language (languages[language]) which returns as capture a Lua table containing data to send to LaTeX.

Here is an example of an item beginning with "Open".
{ "Open" , \begin{uncover}<2>" , \end{cover} }

In order to deal with the ends of lines, we have to close the environment (\begin{cover} in this example) at the end of each line and reopen it at the beginning of the new line. That's why we use two Lua stacks, called left_stack and right_stack. left_stack will be for the elements like \begin{uncover}<2> and right_stack will be for the elements like \end{cover}.

if one_item[1] == "Open"
    then
        tex.sprint( one_item[2] )
        table.insert(left_stack,one_item[2])
        table.insert(right_stack,one_item[3])
    else
        if one_item[1] == "Close"
            then
                tex.sprint( right_stack[#right_stack] )
                left_stack[#left_stack] = nil
                right_stack[#right_stack] = nil
            else
                tex.tprint(one_item)
            end
        end
    end
end
The function \texttt{ParseFile} will be used by the LaTeX command \texttt{\PitonInputFile}. That function merely reads the whole file (that is to say all its lines) and then apply the function \texttt{Parse} to the resulting Lua string.

\begin{verbatim}
function piton.ParseFile(language,name,first_line,last_line)
  local 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

  We extract the BOM of utf-8, if present.
  if string.byte(s,1) == 13 then
    if string.byte(s,2) == 239 then
      if string.byte(s,3) == 187 then
        if string.byte(s,4) == 191 then
          s = string.sub(s,5,-1)
        end
      end
    end
  end

  piton.Parse(language,s)
end
\end{verbatim}

8.3.6 Two variants of the function \texttt{Parse} with integrated preprocessors

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

\begin{verbatim}
function piton.ParseBis(language,code)
  local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
  return piton.Parse(language,s)
end
\end{verbatim}

The following command will be used when we have to parse some small chunks of code that have yet been parsed. They are re-scanned by LaTeX because it has been required by \texttt{\@@_piton:n} in the \texttt{piton} style of the syntactic element. In that case, you have to remove the potential \texttt{\@@_breakable_space} that have been inserted when the key \texttt{break-lines} is in force.

\begin{verbatim}
function piton.ParseTer(language,code)
  local s = ( Cs ( ( P '\@@_breakable_space:' / ' ' + 1 ) ^ 0 ) )
    : match ( code )
  return piton.Parse(language,s)
end
\end{verbatim}

8.3.7 Preprocessors of the function \texttt{Parse} for gobble

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

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

\begin{verbatim}
local function gobble(n,code)
  function concat(acc,new_value)
    return acc .. new_value
  end
  if n==0 then return code end
\end{verbatim}
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 \texttt{add} will be used in the following LPEG \texttt{AutoGobbleLPEG}, \texttt{TabsAutoGobbleLPEG} and \texttt{EnvGobbleLPEG}.

\begin{verbatim}
local function add(acc,new_value)
    return acc + new_value
end
\end{verbatim}

local \texttt{add} will be used in the following LPEG \texttt{AutoGobbleLPEG}, \texttt{TabsAutoGobbleLPEG} and \texttt{EnvGobbleLPEG}.

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 \texttt{\texttt{Cf}}, one using \texttt{add} and the other (encompassing the previous one) using \texttt{math.min} as folding operator.

\begin{verbatim}
local AutoGobbleLPEG =
    ( space ^ 0 * P "\r" ) ^ -1
    * Cf (\(\texttt{( 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).

\(\texttt{( P " " ) ^ 0 * P "\r" +}
\texttt{Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )}
\texttt{*( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,}
\texttt{math.min}
)
\end{verbatim}

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

\begin{verbatim}
* \texttt{( Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )}
  \texttt{*( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,}
  \texttt{math.min}
\end{verbatim}

The following LPEG is similar but works with the indentations.

\begin{verbatim}
local TabsAutoGobbleLPEG =
    ( space ^ 0 * P "\r" ) ^ -1
    * Cf (\(\texttt{( P "\t" ) ^ 0 * P "\r" +}
    \texttt{Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )}
    \texttt{*( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * P "\r"
    ) ^ 0}
    \texttt{*( Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )}
    \texttt{*( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,}
    \texttt{math.min}
\end{verbatim}

The following LPEG returns as capture the number of spaces at the last line, that is to say before the \texttt{\end{Piton}} (and usually it’s also the number of spaces before the corresponding \texttt{\begin{Piton}} because that’s the traditional way to indent in LaTeX). The main work is done by a fold capture \texttt{\texttt{Cf}} using the function \texttt{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(language,n,code)
if n==1
then n = AutoGobbleLPEG : match(code)
else if n==2
then n = EnvGobbleLPEG : match(code)
else if n==3
then n = TabsAutoGobbleLPEG : match(code)
end
end
end

piton.Parse(language,gobble(n,code))
end

8.3.8 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 .. '}'
)
The following function stores in \texttt{\l_@@_first_line_int} and \texttt{\l_@@_last_line_int} the numbers of lines of the file \texttt{file_name} corresponding to the strings \texttt{marker\_beginning} and \texttt{marker\_end}.

\begin{verbatim}
function piton.ComputeRange(marker\_beginning,marker\_end,file\_name)
local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( marker\_beginning )
local t = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( marker\_end )
local first\_line = -1
local count = 0
local last\_found = false
for line in io\_lines(file\_name)
do if first\_line == -1
then if string\_sub(line,1,#s) == s
then first\_line = count
end
else if string\_sub(line,1,#t) == t
then last\_found = true
break
end
end
end
end if first\_line == -1
then tex\_sprint("\PitonBeginMarkerNotFound")
else if last\_found == false
then tex\_sprint("\PitonEndMarkerNotFound")
eendif
end
tex\_sprint(luatexbase\_catcodetables\_expl ,
'\int\_set:Nn \l_@@\_first\_line\_int {\,'.. first\_line .. '} + 2 }'
.. '\int\_set:Nn \l_@@\_last\_line\_int {\,'.. count .. '}')
end
end
\end{verbatim}

\section{History}

The successive versions of the file \texttt{piton.sty} provided by TeXLive are available on the \texttt{svn} server of TeXLive:

\texttt{https://tug.org/svn/texlive/trunk/Master/texmf-dist/tex/lualatex/piton/piton.sty}

The development of the extension piton is done on the following GitHub repository:

\texttt{https://github.com/fpantigny/piton}

Changes between versions 2.0 and 2.1

The key \texttt{line-numbers} has now subkeys \texttt{line-numbers/skip-empty-lines}, \texttt{line-numbers/label-empty-lines}, etc.

The key \texttt{all-line-numbers} is deprecated: use \texttt{line-numbers/skip-empty-lines=false}.

New system to import, with \texttt{\PitonInputFile}, only a part (of the file) delimited by textual markers.

New keys \texttt{begin-escape}, \texttt{end-escape}, \texttt{begin-escape-math} and \texttt{end-escape-math}.

The key \texttt{escape-inside} is deprecated: use \texttt{begin-escape} and \texttt{end-escape}.

Changes between versions 1.6 and 2.0

The extension piton now supports the computer languages OCaml and C (and, of course, Python).
Changes between versions 1.5 and 1.6

New key width (for the total width of the listing).
New style UserFunction to format the names of the Python functions previously defined by the user.
Command \PitonClearUserFunctions to clear the list of such functions names.

Changes between versions 1.4 and 1.5

New key numbers-sep.

Changes between versions 1.3 and 1.4

New key identifiers in \PitonOptions.
New command \PitonStyle.
background-color now accepts as value a list of colors.

Changes between versions 1.2 and 1.3

When the class Beamer is used, the environment \{Piton\} and the command \PitonInputFile are “overlay-aware” (that is to say, they accept a specification of overlays between angular brackets).
New key prompt-background-color
It’s now possible to use the command \label to reference a line of code in an environment \{Piton\}.
A new command \ is available in the argument of the command \piton{...} to insert a space (otherwise, several spaces are replaced by a single space).

Changes between versions 1.1 and 1.2

New key show-spaces-in-string and modification of the key show-spaces.
When the class beamer is used, the environments \{uncoverenv\}, \{onlyenv\}, \{visibleenv\} and \{invisibleenv\}

Changes between versions 1.0 and 1.1

The extension piton detects the class beamer and activates the commands \action, \alert, \invisible, \only, \uncover and \visible in the environments \{Piton\} when the class beamer is used.

Changes between versions 0.99 and 1.0

New key tabs-auto-gobble.

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).

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 0.8 and 0.9

New key tab-size.
Integer value for the key splittable.
Changes between versions 0.7 and 0.8

New keys `footnote` and `footnotehyper` at load-time.
New key `left-margin`.

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`.

Contents

1 Presentation 1

2 Installation 1

3 Use of the package 2
  3.1 Loading the package ........................................ 2
  3.2 Choice of the computer language ............................... 2
  3.3 The tools provided to the user ................................ 2
  3.4 The syntax of the command \piton ............................ 2

4 Customization 3
  4.1 The keys of the command \PitonOptions ...................... 3
  4.2 The styles .................................................. 6
  4.3 Creation of new environments ............................... 7

5 Advanced features 7
  5.1 Page breaks and line breaks ................................ 7
    5.1.1 Page breaks ........................................... 7
    5.1.2 Line breaks ........................................... 8
  5.2 Insertion of a part of a file ................................ 9
    5.2.1 With line numbers .................................... 9
    5.2.2 With textual markers .................................. 9
  5.3 Highlighting some identifiers ............................... 10
  5.4 Mechanisms to escape to LaTeX .............................. 11
    5.4.1 The “LaTeX comments” ................................ 12
    5.4.2 The key “math-comments” ................................ 12
    5.4.3 The mechanism “escape” ................................ 13
    5.4.4 The mechanism “escape-math” .......................... 14
  5.5 Behaviour in the class Beamer .............................. 14
    5.5.1 {Piton} et \PitonInputFile are “overlay-aware” ....... 15
    5.5.2 Commands of Beamer allowed in {Piton} and \PitonInputFile .............................................. 15
    5.5.3 Environments of Beamer allowed in {Piton} and \PitonInputFile .............................................. 16
  5.6 Footnotes in the environments of piton ...................... 16
  5.7 Tabulations .................................................. 17

6 Examples 17
  6.1 Line numbering .............................................. 17
  6.2 Formatting of the LaTeX comments ............................ 17
  6.3 Notes in the listings ....................................... 18
  6.4 An example of tuning of the styles ......................... 19
  6.5 Use with pyluatex .......................................... 20

7 The styles for the different computer languages 22
  7.1 The language Python ........................................ 22
  7.2 The language OCaml .......................................... 23
  7.3 The language C (and C++) ................................... 24
8 Implementation

8.1 Introduction ........................................... 25
8.2 The L3 part of the implementation ..................... 26
  8.2.1 Declaration of the package ........................... 26
  8.2.2 Parameters and technical definitions ................ 28
  8.2.3 Treatment of a line of code ........................... 32
  8.2.4 PitonOptions ........................................ 35
  8.2.5 The numbers of the lines ............................... 39
  8.2.6 The command to write on the aux file ................ 39
  8.2.7 The main commands and environments for the final user ...... 40
  8.2.8 The styles .......................................... 47
  8.2.9 The initial styles ..................................... 48
  8.2.10 Highlighting some identifiers ........................ 49
  8.2.11 Security ........................................... 50
  8.2.12 The error messages of the package .................... 50
  8.2.13 We load piton.lua .................................. 52
8.3 The Lua part of the implementation ...................... 52
  8.3.1 Special functions dealing with LPEG ................... 53
  8.3.2 The LPEG python ..................................... 56
  8.3.3 The LPEG ocaml ..................................... 65
  8.3.4 The LPEG language C .................................. 71
  8.3.5 The function Parse .................................... 76
  8.3.6 Two variants of the function Parse with integrated preprocessors ....... 77
  8.3.7 Preprocessors of the function Parse for gobble ............. 77
  8.3.8 To count the number of lines .......................... 79

9 History ..................................................... 80