

# The package `piton`\*

F. Pantigny  
fpantigny@wanadoo.fr

July 8, 2023

## Abstract

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

## 1 Presentation

The package `piton` uses the Lua library LPEG<sup>1</sup> for parsing Python listings and typeset 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 `--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 `{Piton}`.

```
from math import pi

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

    n is the number of terms in the sum
    """
    if x < 0:
        return -arctan(-x) # recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)
        (we have used that arctan(x) + arctan(1/x) =  $\frac{\pi}{2}$  for  $x > 0$ )2
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

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

---

\*This document corresponds to the version 1.6a of `piton`, at the date of 2023/07/08.

<sup>1</sup>LPEG is a pattern-matching library for Lua, written in C, based on *parsing expression grammars*: <http://www.inf.puc-rio.br/~roberto/lpeg/>

<sup>2</sup>This LaTeX escape has been done by beginning the comment by `#>`.

## 2 Use of the package

### 2.1 Loading the package

The package `piton` should be loaded with the classical command `\usepackage{piton}`. Nevertheless, we have two remarks:

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

### 2.2 The tools provided to the user

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

- The command `\piton` should be used to typeset small pieces of code inside a paragraph. For example:

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

The syntax and particularities of the command `\piton` are detailed below.

- The environment `{Piton}` should be used to typeset multi-lines code. Since it takes its argument in a verbatim mode, it can't be used within the argument of a LaTeX command. For sake of customization, it's possible to define new environments similar to the environment `{Piton}` with the command `\NewPitonEnvironment`: cf. 3.3 p. 6.
- The command `\PitonInputFile` is used to insert and typeset a whole external file.

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

### 2.3 The syntax of the command `\piton`

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

- **Syntax `\piton{...}`**

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

- several consecutive spaces will be replaced by only one space,  
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<sup>3</sup> are fully expanded and not executed,  
so it's possible to use `\\` to insert a backslash.

---

<sup>3</sup>That concerns the commands beginning with a backslash but also the active characters (with catcode equal to 13).

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

Examples :

<code>\piton{MyString = '\n'}</code>	<code>MyString = '\n'</code>
<code>\piton{def even(n): return n%2==0}</code>	<code>def even(n): return n%2==0</code>
<code>\piton{c="#" # an affectation }</code>	<code>c="#" # an affectation</code>
<code>\piton{c="#" \ \ \ # an affectation }</code>	<code>c="#" # an affectation</code>
<code>\piton{MyDict = {'a': 3, 'b': 4 }}</code>	<code>MyDict = {'a': 3, 'b': 4}</code>

It's possible to use the command `\piton` in the arguments of a LaTeX command.<sup>4</sup>

- **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 :

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

## 3 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`.

### 3.1 The command `\PitonOptions`

The command `\PitonOptions` takes in as argument a comma-separated list of *key=value* pairs. The scope of the settings done by that command is the current TeX group.<sup>5</sup>

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

---

<sup>4</sup>For example, it's possible to use the command `\piton` in a footnote. Example : `s = 'A string'`.

<sup>5</sup>We remind that a LaTeX environment is, in particular, a TeX group.

- **New 1.5**

The key `numbers-sep` is the horizontal distance between the numbers of lines (inserted by `line-numbers` or `all-line-numbers`) and the beginning of the lines of code. The initial value is 0.7 em.

- With the key `resume`, the counter of lines is not set to zero at the beginning of each environment `{Piton}` or use of `\PitonInputFile` as it is otherwise. That allows a numbering of the lines across several environments.
- The key `left-margin` corresponds to a margin on the left. That key may be useful in conjunction with the key `line-numbers` or the key `line-all-numbers` if one does not want the numbers in an overlapping position on the left.

It's possible to use the key `left-margin` with the value `auto`. With that value, if the key `line-numbers` or the key `all-line-numbers` is used, a margin will be automatically inserted to fit the numbers of lines. See an example part 5.1 on page 14.

- 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}`.

- With the key `prompt-background-color`, `piton` adds a color background to the lines beginning with the prompt `>>>` (and its continuation `...`) characteristic of the Python consoles with REPL (*read-eval-print loop*).

- **New 1.6** The key `width` will fix the width of the listing. That width applies to the colored backgrounds specified by `background-color` and `prompt-background-color` but also for the automatic breaking of the lines (when required by `break-lines`: cf. 4.4.2, p. 13).

That key may take in as value a numeric value but also the special value `min`. With that value, the width will be computed from the maximal width of the lines of code. Caution: the special value `min` requires two compilations with LuaLaTeX<sup>6</sup>.

For an example of use of `width=min`, see the section 5.2, p. 15.

- When the key `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 `□` (U+2423 : OPEN BOX). Of course, that character U+2423 must be present in the monospaced font which is used.<sup>7</sup>

*Example :* `my_string = 'Very□good□answer'`

With the key `show-spaces`, all the spaces are replaced by U+2423 (and no line break can occur on those “visible spaces”, even when the key `break-lines`<sup>8</sup> is in force).

```
\PitonOptions{line-numbers,auto-gobble,background-color = gray!15}
\begin{Piton}
  from math import pi
  def arctan(x,n=10):
    """Compute the mathematical value of arctan(x)
```

<sup>6</sup>The maximal width is computed during the first compilation, written on the `aux` file and re-used during the second compilation. Several tools such as `latexmk` (used by Overleaf) do automatically a sufficient number of compilations.

<sup>7</sup>The package `piton` simply uses the current monospaced font. The best way to change that font is to use the command `\setmonofont` of the package `fontspec`.

<sup>8</sup>cf. 4.4.2 p. 13

```

n is the number of terms in the sum
"""
if x < 0:
    return -arctan(-x) # recursive call
elif x > 1:
    return pi/2 - arctan(1/x)
    #> (we have used that  $\arctan(x) + \arctan(1/x) = \frac{\pi}{2}$  pour  $x > 0$ )
else
    s = 0
    for k in range(n):
        s += (-1)**k/(2*k+1)*x**(2*k+1)
    return s
\end{Piton}

```

```

1  from math import pi
2  def arctan(x,n=10):
3      """Compute the mathematical value of arctan(x)
4
5      n is the number of terms in the sum
6      """
7      if x < 0:
8          return -arctan(-x) # recursive call
9      elif x > 1:
10         return pi/2 - arctan(1/x)
11         (we have used that  $\arctan(x) + \arctan(1/x) = \frac{\pi}{2}$  for  $x > 0$ )
12     else
13         s = 0
14         for k in range(n):
15             s += (-1)**k/(2*k+1)*x**(2*k+1)
16         return s

```

The command `\PitonOptions` provides in fact several other keys which will be described further (see in particular the “Pages breaks and line breaks” p. 12).

## 3.2 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.<sup>9</sup>

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

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`).

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

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

---

<sup>9</sup>We remind that a LaTeX environment is, in particular, a TeX group.

The different styles are described in the table 1. The initial settings done by `piton` in `piton.sty` are inspired by the style `manni` de Pygments.<sup>10</sup>

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`.

### New 1.6

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 `\PitonSetStyle`.

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*).

```
\PitonSetStyle{UserFunction = \color[HTML]{CC00FF}}
```

```
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 `transpse` 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 défined is kept in the memory of LuaLaTeX (in a global way, that is to say independtly of the TeX groups). The extension `piton` provides a command to clear that list : it's the command `\PitonClearUserFunctions`.

## 3.3 Creation of new environments

Since the environment `{Piton}` has to catch its body in a special way (more or less as verbatim text), it's not possible to construct new environments directly over the environment `{Piton}` with the classical commands `\newenvironment` or `\NewDocumentEnvironment`.

That's why `piton` provides a command `\NewPitonEnvironment`. That command takes in three mandatory arguments.

That command has the same syntax as the classical environment `\NewDocumentEnvironment`.

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

```
\NewPitonEnvironment{Python}{-}{-}{-}
```

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

<sup>10</sup>See: <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 `{Piton}` with the instruction `\PitonOptions{background-color = [HTML]{F0F3F3}}`.

```
\NewPitonEnvironment{Python}{0{}}{\PitonOptions{#1}}{}
```

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

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

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}
```

```
def square(x):
    """Compute the square of a number"""
    return x*x
```

## 4 Advanced features

### 4.1 Highlighting some identifiers

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

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

```
{ names = names, style = instructions }
```

- *names* is a (comma-separated) list of identifiers names;
- *instructions* is a list of LaTeX instructions of the same type that `piton` “styles” previously presented (cf 3.2 p. 5).

*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 `\textsl{\ttfamily names}`.

```
\PitonOptions
{
    identifiers =
    {
        names = { l1 , l2 } ,
        style = \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}

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

## 4.2 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. 4.3 p. 10.

### 4.2.1 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 syntactic mark (which, by default, is `#>`). For this purpose, there is a key `comment-latex` available at load-time (that is to say at the `\usepackage`) which allows to choose the characters which, preceded by `#`, will be the syntactic marker.

For example, with the following loading:



```
\usepackage[comment-latex = LaTeX]{piton}
```

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:

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

For other examples of customization of the LaTeX comments, see the part 5.2 p. 15

If the user has required line numbers in the left margin (with the key `line-numbers` or the key `all-line-numbers` of `\PitonOptions`), it’s possible to refer to a number of line with the command `\label` used in a LaTeX comment.<sup>11</sup>

#### 4.2.2 The key “math-comments”

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

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

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

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

```
def square(x):
    return x*x # compute  $x^2$ 
```

#### 4.2.3 The mechanism “escape-inside”

It’s also possible to overwrite the Python listings to insert LaTeX code almost everywhere (but between lexical units, of course). By default, `piton` does not fix any character for that kind of escape. In order to use this mechanism, it’s necessary to specify two characters which will delimit the escape (one for the beginning and one for the end) by using the key `escape-inside` at load-time (that is to say at the `\begin{documnt}`).

In the following example, we assume that the extension `piton` has been loaded by the following instruction.

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

In the following code, which is a recursive programming 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`).

---

<sup>11</sup>That 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.)

```

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

### 4.3 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]`.<sup>12</sup>

When the package `piton` is used within the class `beamer`<sup>13</sup>, the behaviour of `piton` is slightly modified, as described now.

<sup>12</sup>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).

<sup>13</sup>The extension `piton` detects the class `beamer` but, if needed, it's also possible to activate that mechanism with the key `beamer` provided by `piton` at load-time: `\usepackage[beamer]{piton}`

### 4.3.1 `{Piton}` et `\PitonInputFile` are “overlay-aware”

When `piton` is used in the class `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}
```

### 4.3.2 Commands of Beamer allowed in `{Piton}` and `\PitonInputFile`

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

- no mandatory argument : `\pause`<sup>14</sup> ;
- one mandatory argument : `\action`, `\alert`, `\invisible`, `\only`, `\uncover` and `\visible` ;
- two mandatory arguments : `\alt` ;
- three mandatory arguments : `\temporal`.

In the mandatory arguments of these commands, the braces must be balanced. However, the braces included in short strings<sup>15</sup> of Python are not considered.

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

Here is a complete example of file:

```
\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->{s = "{" + str(l[0])}
    \only<3->{for x in l[1:]: s = s + "," + str(x)}
    \only<4->{s = s + "}"}
    return s
\end{Piton}
\end{frame}
\end{document}
```

In the previous example, the braces in the Python strings `"{"` and `"}"` are correctly interpreted (without any escape character).

---

<sup>14</sup>One should remark that it’s also possible to use the command `\pause` in a “LaTeX comment”, that is to say by writing `#> \pause`. By this way, if the Python code is copied, it’s still executable by Python

<sup>15</sup>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.

### 4.3.3 Environments of Beamer allowed in `{Piton}` and `\PitonInputFile`

When `piton` is used in the class `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 `\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 `\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 `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 `\@highLight` of `lua-ul` (that extension requires also the package `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 `\alert` relies upon that environment `{alertenv}`).

## 4.4 Page breaks and line breaks

### 4.4.1 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.<sup>16</sup>

#### 4.4.2 Line breaks

By default, the elements produced by `piton` can't be broken by an end of 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: `\hspace*{0.5em}\textbackslash`.
- The key `continuation-symbol` corresponds to the symbol placed at each carriage return. The initial value is: `+;`.
- 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}
```

```
def dict_of_list(l):
    """Converts a list of subrs and descriptions of glyphs in \
    ↪ 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 \
        ↪ list_letter[1:-1]]
    return dict
```

<sup>16</sup>With the key `splittable`, the environments `{Piton}` are breakable, even within a (breakable) environment of `tclobox`. Remind that an environment of `tclobox` included in another environment of `tclobox` is *not* breakable, even when both environments use the key `breakable` of `tclobox`.

## 4.5 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 `\footnotemark`–`\footnotetext`.

However, it's also possible to extract the footnotes with the help of the package `footnote` or the package `footnotehyper`.

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

If `piton` is loaded with the option `footnotehyper`, the package `footnotehyper` is loaded (if it is not yet loaded) and it is used to extract footnotes.

Caution: The packages `footnote` and `footnotehyper` are incompatible. The package `footnotehyper` is the successor of the package `footnote` and should be used preferently. The package `footnote` has some drawbacks, in particular: it must be loaded after the package `xcolor` and it is not perfectly compatible with `hyperref`.

In this document, the package `piton` has been loaded with the option `footnotehyper`. For examples of notes, cf. 5.3, p. 16.

## 4.6 Tabulations

Even though it's recommended to indent the Python listings with spaces (see PEP 8), `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 `tab-size` of `\PitonOptions`.

There exists also a key `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 `{Piton}` and applies `gobble` with that value of  $n$  (before replacement of the tabulations by spaces, of course). Hence, that key is similar to the key `auto-gobble` but acts on U+0009 instead of U+0020 (spaces).

# 5 Examples

## 5.1 Line numbering

We remind that it's possible to have an automatic numbering of the lines in the Python listings by using the key `line-numbers` or the key `all-line-numbers`.

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

In order to avoid that overlapping, it's possible to use the option `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)          #> (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}
```

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

## 5.2 Formatting of the LaTeX comments

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

```
\PitonOptions{background-color=gray!10}
\SetPitonStyle{Comment.LaTeX = \hfill \normalfont\color{gray}}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)          #> 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:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

It's also possible to display these LaTeX comments in a kind of second column by limiting the width of the Python code with the key `width`. In the following example, we use the key `width` with the special value `min`.

```
\PitonOptions{background-color=gray!10, width=min}
\NewDocumentCommand{\MyLaTeXCommand}{m}{\hfill \normalfont\itshape\rlap{\quad #1}}
\SetPitonStyle{Comment.LaTeX = \MyLaTeXCommand}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x) #> 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}
```

```
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
```

### 5.3 Notes in the listings

In order to be able to extract the notes (which are typeset with the command `\footnote`), the extension `piton` must be loaded with the key `footnote` or the key `footnotehyper` as explained in the section 4.5 p. 14. In this document, the extension `piton` has been loaded with the key `footnotehyper`. Of course, in an environment `{Piton}`, a command `\footnote` may appear only within a LaTeX comment (which begins with `#>`). It's possible to have comments which contain only that command `\footnote`. That's the case in the following example.

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

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)17
    elif x > 1:
        return pi/2 - arctan(1/x)18
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

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

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

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

---

<sup>a</sup>First recursive call.

<sup>b</sup>Second recursive call.

---

<sup>17</sup>First recursive call.

<sup>18</sup>Second recursive call.



## 5.4 An example of tuning of the styles

The graphical styles have been presented in the section 3.2, p. 5.

We present now an example of tuning of these styles adapted to the documents in black and white. We use the font *Deja Vu Sans Mono*<sup>19</sup> specified by the command `\setmonofont` of `fontspec`.

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

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

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

```
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
```

## 5.5 Use with pyluatex

The package `pyluatex` is an extension which allows the execution of some Python code from `lualatex` (provided that Python is installed on the machine and that the compilation is done with `lualatex` and `--shell-escape`).

Here is, for example, an environment `{PitonExecute}` which formats a Python listing (with `piton`) but display also the output of the execution of the code with Python (for technical reasons, the `!` is mandatory in the signature of the environment).

---

<sup>19</sup>See: <https://dejavu-fonts.github.io>

```

\ExplSyntaxOn
\NewDocumentEnvironment { PitonExecute } { ! O { } } % the ! is mandatory
{
  \PyLTVerbatimEnv
  \begin{pythonq}
}
{
  \end{pythonq}
  \directlua
  {
    tex.print("\PitonOptions{#1}")
    tex.print("\begin{Piton}")
    tex.print(pyluatex.get_last_code())
    tex.print("\end{Piton}")
    tex.print("")
  }
  \begin{center}
    \directlua{tex.print(pyluatex.get_last_output())}
  \end{center}
}
\ExplSyntaxOff

```

This environment `{PitonExecute}` takes in as optional argument (between square brackets) the options of the command `\PitonOptions`.

**Table 1:** Usage of the different styles

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

## 6 Implementation

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

### 6.1 Introduction

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

In fact, all that job is done by a LPEG called `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.<sup>20</sup>

Consider, for example, the following Python code:

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

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

```
{ "\\_piton_begin_line:" }a  
{ "{\\PitonStyle{Keyword}{ " } }b  
{ luatexbase.catcodetables.CatcodeTableOtherc, "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:" }
```

---

<sup>a</sup>Each line of the Python listings will be encapsulated in a pair: `\\_begin_line: – \\_end_line:`. The token `\\_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\\_begin_line:`. Both tokens `\\_begin_line:` and `\\_end_line:` will be nullified in the command `\\piton` (since there can't be lines breaks in the argument of a command `\\piton`).

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

<sup>c</sup>`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).

---

<sup>20</sup>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}}
\_{\PitonStyle{Operator}{%}}{\PitonStyle{Number}{2}}\__piton_end_line:

```

## 6.2 The L3 part of the implementation

### 6.2.1 Declaration of the package

```

1 \NeedsTeXFormat{LaTeX2e}
2 \RequirePackage{l3keys2e}
3 \ProvidesExplPackage
4   {piton}
5   {\myfiledate}
6   {\myfileversion}
7   {Highlight Python codes with LPEG on LuaLaTeX}

8 \msg_new:nnn { piton } { LuaLaTeX-mandatory }
9   {
10     LuaLaTeX~is~mandatory.\\
11     The~package~'piton'~requires~the~engine~LuaLaTeX.\\
12     \str_if_eq:VnT \c_sys_jobname_str { output }
13       { If~you~use~Overleaf,~you~can~switch~to~LuaLaTeX~in~the~"Menu". \\}
14     If~you~go~on,~the~package~'piton'~won't~be~loaded.
15   }
16 \sys_if_engine luatex:F { \msg_critical:nn { piton } { LuaLaTeX-mandatory } }

17 \RequirePackage { luatexbase }

```

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

```
18 \bool_new:N \c_@@_footnotehyper_bool
```

The boolean `\c_@@_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.

```
19 \bool_new:N \c_@@_footnote_bool
```

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

```
20 \bool_new:N \c_@@_math_comments_bool
```

The following boolean corresponds to the key `beamer`.

```
21 \bool_new:N \c_@@_beamer_bool
```

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

```

22 \keys_define:nn { piton / package }
23   {
24     footnote .bool_set:N = \c_@@_footnote_bool ,
25     footnotehyper .bool_set:N = \c_@@_footnotehyper_bool ,
26     escape-inside .tl_set:N = \c_@@_escape_inside_tl ,
27     escape-inside .initial:n = ,
28     comment-latex .code:n = { \lua_now:n { comment_latex = "#1" } } ,
29     comment-latex .value_required:n = true ,
30     math-comments .bool_set:N = \c_@@_math_comments_bool ,
31     math-comments .default:n = true ,
32     beamer .bool_set:N = \c_@@_beamer_bool ,
33     beamer .default:n = true ,
34     unknown .code:n = \msg_error:nn { piton } { unknown-key-for-package }

```

```

35 }
36 \msg_new:nnn { piton } { unknown-key-for-package }
37 {
38   Unknown-key.\
39   You-have-used-the-key~'\l_keys_key_str'~but-the-only-keys-available-here~
40   are~'beamer',~'comment-latex',~'escape-inside',~'footnote',~'footnotehyper'~and~
41   'math-comments'.~Other-keys-are-available-in~\token_to_str:N \PitonOptions.\
42   That-key-will-be-ignored.
43 }

```

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

```

44 \ProcessKeysOptions { piton / package }

45 \beginingroup
46 \cs_new_protected:Npn \@@_set_escape_char:nn #1 #2
47 {
48   \lua_now:n { piton_begin_escape = "#1" }
49   \lua_now:n { piton_end_escape = "#2" }
50 }
51 \cs_generate_variant:Nn \@@_set_escape_char:nn { x x }
52 \@@_set_escape_char:xx
53 { \tl_head:V \c_@@_escape_inside_tl }
54 { \tl_tail:V \c_@@_escape_inside_tl }
55 \endgroup

56 \@ifclassloaded { beamer } { \bool_set_true:N \c_@@_beamer_bool } { }
57 \bool_if:NT \c_@@_beamer_bool { \lua_now:n { piton_beamer = true } }

58 \hook_gput_code:nnn { begindocument } { . }
59 {
60   \@ifpackageloaded { xcolor }
61   { }
62   { \msg_fatal:nn { piton } { xcolor~not~loaded } }
63 }

64 \msg_new:nnn { piton } { xcolor~not~loaded }
65 {
66   xcolor~not~loaded \
67   The~package~'xcolor'~is~required~by~'piton'.\
68   This~error~is~fatal.
69 }

70 \msg_new:nnn { piton } { footnote~with~footnotehyper~package }
71 {
72   Footnote~forbidden.\
73   You~can't~use~the~option~'footnote'~because~the~package~
74   footnotehyper~has~already~been~loaded.~
75   If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
76   within~the~environments~of~piton~will~be~extracted~with~the~tools~
77   of~the~package~footnotehyper.\
78   If~you~go~on,~the~package~footnote~won't~be~loaded.
79 }

80 \msg_new:nnn { piton } { footnotehyper~with~footnote~package }
81 {
82   You~can't~use~the~option~'footnotehyper'~because~the~package~
83   footnote~has~already~been~loaded.~
84   If~you~want,~you~can~use~the~option~'footnote'~and~the~footnotes~
85   within~the~environments~of~piton~will~be~extracted~with~the~tools~
86   of~the~package~footnote.\
87   If~you~go~on,~the~package~footnotehyper~won't~be~loaded.
88 }

```

```

89 \bool_if:NT \c_@@_footnote_bool
90 {

```

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

```

91 \ifclassloaded { beamer }
92 { \bool_set_false:N \c_@@_footnote_bool }
93 {
94 \ifpackageloaded { footnotehyper }
95 { \@@_error:n { footnote-with-footnotehyper-package } }
96 { \usepackage { footnote } }
97 }
98 }
99 \bool_if:NT \c_@@_footnotehyper_bool
100 {

```

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

```

101 \ifclassloaded { beamer }
102 { \bool_set_false:N \c_@@_footnote_bool }
103 {
104 \ifpackageloaded { footnote }
105 { \@@_error:n { footnotehyper-with-footnote-package } }
106 { \usepackage { footnotehyper } }
107 \bool_set_true:N \c_@@_footnote_bool
108 }
109 }

```

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

### 6.2.2 Parameters and technical definitions

The following string will contain the name of the informatic language considered (the initial value is `python`).

```

110 \str_new:N \l_@@_language_str
111 \str_set:Nn \l_@@_language_str { python }

```

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

```

112 \int_new:N \l_@@_nb_lines_int

```

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

```

113 \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.

```

114 \int_new:N \g_@@_line_int

```

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

```

115 \tl_new:N \g_@@_aux_tl

```

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

```

116 \int_new:N \l_@@_splittable_int

```

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

```

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

```

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

```

118 \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`).

```
119 \tl_new:N \l_@@_prompt_bg_color_tl
```

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

```
120 \int_new:N \g_@@_env_int
```

The following boolean corresponds to the key `show-spaces`.

```
121 \bool_new:N \l_@@_show_spaces_bool
```

The following booleans correspond to the keys `break-lines` and `indent-broken-lines`.

```
122 \bool_new:N \l_@@_break_lines_in_Piton_bool
```

```
123 \bool_new:N \l_@@_indent_broken_lines_bool
```

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

```
124 \tl_new:N \l_@@_continuation_symbol_tl
```

```
125 \tl_set:Nn \l_@@_continuation_symbol_tl { + }
```

```
126 % The following token list corresponds to the key
```

```
127 % |continuation-symbol-on-indentation|. The name has been shorten to |csoi|.
```

```
128 \tl_new:N \l_@@_csoi_tl
```

```
129 \tl_set:Nn \l_@@_csoi_tl { $ \hookrightarrow \; $ }
```

The following token list corresponds to the key `end-of-broken-line`.

```
130 \tl_new:N \l_@@_end_of_broken_line_tl
```

```
131 \tl_set:Nn \l_@@_end_of_broken_line_tl { \hspace*{0.5em} \textbackslash }
```

The following boolean corresponds to the key `break-lines-in-piton`.

```
132 \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 `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`.

```
133 \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`).

```
134 \dim_new:N \l_@@_line_width_dim
```

The following flag will be raised with the key `width` is used with the special value `min`.

```
135 \bool_new:N \l_@@_width_min_bool
```



If the key `width` is used with the special value `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 `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.

```
136 \dim_new:N \g_@@_tmp_width_dim
```

The following dimension corresponds to the key `left-margin` of `\PitonOptions`.

```
137 \dim_new:N \l_@@_left_margin_dim
```

The following boolean will be set when the key `left-margin=auto` is used.

```
138 \bool_new:N \l_@@_left_margin_auto_bool
```

The following dimension corresponds to the key `numbers-sep` of `\PitonOptions`.

```
139 \dim_new:N \l_@@_numbers_sep_dim
140 \dim_set:Nn \l_@@_numbers_sep_dim { 0.7 em }
```

The tabulators will be replaced by the content of the following token list.

```
141 \tl_new:N \l_@@_tab_tl

142 \cs_new_protected:Npn \@@_set_tab_tl:n #1
143 {
144   \tl_clear:N \l_@@_tab_tl
145   \prg_replicate:nn { #1 }
146     { \tl_put_right:Nn \l_@@_tab_tl { ~ } }
147 }
148 \@@_set_tab_tl:n { 4 }
```

The following integer corresponds to the key `gobble`.

```
149 \int_new:N \l_@@_gobble_int

150 \tl_new:N \l_@@_space_tl
151 \tl_set:Nn \l_@@_space_tl { ~ }
```

At each line, the following counter will count the spaces at the beginning.

```
152 \int_new:N \g_@@_indentation_int

153 \cs_new_protected:Npn \@@_an_indentation_space:
154 { \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.

```
155 \cs_new_protected:Npn \@@_beamer_command:n #1
156 {
157   \str_set:Nn \l_@@_beamer_command_str { #1 }
158   \use:c { #1 }
159 }
```

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

```
160 \cs_new_protected:Npn \@@_label:n #1
161 {
162   \bool_if:NTF \l_@@_line_numbers_bool
163     {
164       \@bsphack
165       \protected@write \@auxout { }
166         {
167           \string \newlabel { #1 }
168           {
```

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

```

169         { \int_eval:n { \g_@@_visual_line_int + 1 } }
170         { \thepage }
171     }
172 }
173 \@@esphack
174 }
175 { \msg_error:nn { piton } { label-with-lines-numbers } }
176 }

```

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

```

177 \cs_new_protected:Npn \@@_open_brace:
178 { \directlua { piton.open_brace() } }
179 \cs_new_protected:Npn \@@_close_brace:
180 { \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.

```

181 \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.

```

182 \cs_new_protected:Npn \@@_prompt:
183 {
184     \tl_gset:Nn \g_@@_begin_line_hook_tl
185     {
186         \tl_if_empty:NF \l_@@_prompt_bg_color_tl % added 2023-04-24
187         { \clist_set:NV \l_@@_bg_color_clist \l_@@_prompt_bg_color_tl }
188     }
189 }

```

### 6.2.3 Treatment of a line of code

```

190 \cs_new_protected:Npn \@@_replace_spaces:n #1
191 {
192     \tl_set:Nn \l_tmpa_tl { #1 }
193     \bool_if:NTF \l_@@_show_spaces_bool
194     { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
195     {

```

If the key `break-lines-in-Piton` is in force, we replace all the characters U+0020 (that is to say the spaces) by \@@\_breakable\_space:. Remark that, except the spaces inserted in the LaTeX comments (and maybe in the math comments), all these spaces are of catcode “other” (=12) and are unbreakable.

```

196     \bool_if:NT \l_@@_break_lines_in_Piton_bool
197     {
198         \regex_replace_all:nnN
199         { \x20 }
200         { \c { @@_breakable_space: } }
201         \l_tmpa_tl
202     }
203 }
204 \l_tmpa_tl
205 }
206 \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.

```

207 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
208 {
209   \group_begin:
210   \g_@@_begin_line_hook_tl
211   \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 currying in the following code.

```

212   \bool_if:NTF \l_@@_width_min_bool
213     \@@_put_in_coffin_ii:n
214     \@@_put_in_coffin_i:n
215   {
216     \language = -1
217     \raggedright
218     \strut
219     \@@_replace_spaces:n { #1 }
220     \strut \hfil
221   }

```

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

```

222   \hbox_set:Nn \l_tmpa_box
223   {
224     \skip_horizontal:N \l_@@_left_margin_dim
225     \bool_if:NT \l_@@_line_numbers_bool
226     {
227       \bool_if:NF \l_@@_all_line_numbers_bool
228       { \tl_if_eq:nnF { #1 } { \PitonStyle {Prompt}}{} } }

```

Remember that `\@@_print_number:` always uses `\hbox_overlap_left:n`.

```

229     \@@_print_number:
230   }

```

If there is a background, we must remind that there is a left margin of 0.5 em for the background...

```

231     \clist_if_empty:NF \l_@@_bg_color_clist
232     {

```

... but if only if the key `left-margin` is not used !

```

233       \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
234       { \skip_horizontal:n { 0.5 em } }
235     }
236     \coffin_typeset:Nnnnn \l_tmpa_coffin T l \c_zero_dim \c_zero_dim
237   }
238   \box_set_dp:Nn \l_tmpa_box { \box_dp:N \l_tmpa_box + 1.25 pt }
239   \box_set_ht:Nn \l_tmpa_box { \box_ht:N \l_tmpa_box + 1.25 pt }
240   \clist_if_empty:NTF \l_@@_bg_color_clist
241     { \box_use_drop:N \l_tmpa_box }
242     {
243       \vtop
244       {
245         \hbox:n
246         {
247           \@@_color:N \l_@@_bg_color_clist
248           \vrule height \box_ht:N \l_tmpa_box
249             depth \box_dp:N \l_tmpa_box
250             width \l_@@_width_dim
251         }
252         \skip_vertical:n { - \box_ht_plus_dp:N \l_tmpa_box }
253         \box_use_drop:N \l_tmpa_box
254       }
255     }
256   \vspace { - 2.5 pt }
257   \group_end:

```

```

258 \tl_gclear:N \g_@@_begin_line_hook_tl
259 }

```

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

That commands takes in its argument by currying.

```

260 \cs_set_protected:Npn \@@_put_in_coffin_i:n
261 { \vcoffin_set:Nnn \l_tmpa_coffin \l_@@_line_width_dim }

```

The second case is the case when the key `width` is used with the special value `min`.

```

262 \cs_set_protected:Npn \@@_put_in_coffin_ii:n #1
263 {

```

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`).

```

264 \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).

```

265 \dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_tmp_width_dim
266 { \dim_gset:Nn \g_@@_tmp_width_dim { \box_wd:N \l_tmpa_box } }
267 \hcoffin_set:Nn \l_tmpa_coffin
268 {
269 \hbox_to_wd:nn \l_@@_line_width_dim

```

We unpack the bock in order to free the potential `\hfill` springs present in the LaTeX comments (cf. section 5.2, p. 15).

```

270 { \hbox_unpack:N \l_tmpa_box \hfil }
271 }
272 }

```

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).

```

273 \cs_set_protected:Npn \@@_color:N #1
274 {
275 \int_set:Nn \l_tmpa_int { \clist_count:N #1 }
276 \int_set:Nn \l_tmpb_int { \int_mod:nn \g_@@_line_int \l_tmpa_int + 1 }
277 \tl_set:Nx \l_tmpa_tl { \clist_item:Nn #1 \l_tmpb_int }
278 \tl_if_eq:NnTF \l_tmpa_tl { none }

```

By setting `\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).

```

279 { \dim_zero:N \l_@@_width_dim }
280 { \exp_args:NV \@@_color_i:n \l_tmpa_tl }
281 }

```

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} }`.

```

282 \cs_set_protected:Npn \@@_color_i:n #1
283 {
284 \tl_if_head_eq_meaning:nNTF { #1 } [
285 {
286 \tl_set:Nn \l_tmpa_tl { #1 }
287 \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
288 \exp_last_unbraced:NV \color \l_tmpa_tl
289 }
290 { \color { #1 } }
291 }
292 \cs_generate_variant:Nn \@@_color:n { V }

```

```

293 \cs_new_protected:Npn \@@_newline:

```

```

294 {
295   \int_gincr:N \g_@@_line_int
296   \int_compare:nNnT \g_@@_line_int > { \l_@@_splittable_int - 1 }
297   {
298     \int_compare:nNnT
299     { \l_@@_nb_lines_int - \g_@@_line_int } > \l_@@_splittable_int
300     {
301       \egroup
302       \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
303       \par \mode_leave_vertical: % \newline
304       \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
305       \vtop \bgroup
306     }
307   }
308 }

309 \cs_set_protected:Npn \@@_breakable_space:
310 {
311   \discretionary
312   { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
313   {
314     \hbox_overlap_left:n
315     {
316       {
317         \normalfont \footnotesize \color { gray }
318         \l_@@_continuation_symbol_tl
319       }
320       \skip_horizontal:n { 0.3 em }
321       \clist_if_empty:NF \l_@@_bg_color_clist
322       { \skip_horizontal:n { 0.5 em } }
323     }
324     \bool_if:NT \l_@@_indent_broken_lines_bool
325     {
326       \hbox:n
327       {
328         \prg_replicate:nn { \g_@@_indentation_int } { ~ }
329         { \color { gray } \l_@@_csoi_tl }
330       }
331     }
332   }
333   { \hbox { ~ } }
334 }

```

## 6.2.4 PitonOptions

The following parameters correspond to the keys `line-numbers` and `all-line-numbers`.

```

335 \bool_new:N \l_@@_line_numbers_bool
336 \bool_new:N \l_@@_all_line_numbers_bool

```

The following flag corresponds to the key `resume`.

```

337 \bool_new:N \l_@@_resume_bool

```

Be careful! The name of the following set of keys must be considered as public! Hence, it should *not* be changed.

```

338 \keys_define:nn { PitonOptions }
339 {
340   language      .str_set:N          = \l_@@_language_str ,
341   language      .value_required:n   = true ,
342   gobble        .int_set:N          = \l_@@_gobble_int ,
343   gobble        .value_required:n   = true ,
344   auto-gobble   .code:n             = \int_set:Nn \l_@@_gobble_int { -1 } ,

```

```

345 auto-gobble      .value_forbidden:n = true ,
346 env-gobble      .code:n             = \int_set:Nn \l_@@_gobble_int { -2 } ,
347 env-gobble      .value_forbidden:n = true ,
348 tabs-auto-gobble .code:n             = \int_set:Nn \l_@@_gobble_int { -3 } ,
349 tabs-auto-gobble .value_forbidden:n = true ,
350 line-numbers     .bool_set:N         = \l_@@_line_numbers_bool ,
351 line-numbers     .default:n         = true ,
352 all-line-numbers .code:n =
353   \bool_set_true:N \l_@@_line_numbers_bool
354   \bool_set_true:N \l_@@_all_line_numbers_bool ,
355 all-line-numbers .value_forbidden:n = true ,
356 resume          .bool_set:N         = \l_@@_resume_bool ,
357 resume          .value_forbidden:n = true ,
358 splittable      .int_set:N          = \l_@@_splittable_int ,
359 splittable      .default:n          = 1 ,
360 background-color .clist_set:N        = \l_@@_bg_color_clist ,
361 background-color .value_required:n = true ,
362 prompt-background-color .tl_set:N    = \l_@@_prompt_bg_color_tl ,
363 prompt-background-color .value_required:n = true ,
364 width           .code:n =
365   \str_if_eq:nnTF { #1 } { min }
366   {
367     \bool_set_true:N \l_@@_width_min_bool
368     \dim_zero:N \l_@@_width_dim
369   }
370   {
371     \bool_set_false:N \l_@@_width_min_bool
372     \dim_set:Nn \l_@@_width_dim { #1 }
373   } ,
374 width           .value_required:n = true ,
375 left-margin     .code:n =
376   \str_if_eq:nnTF { #1 } { auto }
377   {
378     \dim_zero:N \l_@@_left_margin_dim
379     \bool_set_true:N \l_@@_left_margin_auto_bool
380   }
381   {
382     \dim_set:Nn \l_@@_left_margin_dim { #1 }
383     \bool_set_false:N \l_@@_left_margin_auto_bool
384   } ,
385 left-margin     .value_required:n = true ,
386 numbers-sep     .dim_set:N         = \l_@@_numbers_sep_dim ,
387 numbers-sep     .value_required:n = true ,
388 tab-size        .code:n             = \@@_set_tab_tl:n { #1 } ,
389 tab-size        .value_required:n = true ,
390 show-spaces     .bool_set:N         = \l_@@_show_spaces_bool ,
391 show-spaces     .default:n         = true ,
392 show-spaces-in-strings .code:n      = \tl_set:Nn \l_@@_space_tl { \_ } , % U+2423
393 show-spaces-in-strings .value_forbidden:n = true ,
394 break-lines-in-Piton .bool_set:N    = \l_@@_break_lines_in_Piton_bool ,
395 break-lines-in-Piton .default:n     = true ,
396 break-lines-in-piton .bool_set:N    = \l_@@_break_lines_in_piton_bool ,
397 break-lines-in-piton .default:n     = true ,
398 break-lines     .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
399 break-lines     .value_forbidden:n = true ,
400 indent-broken-lines .bool_set:N     = \l_@@_indent_broken_lines_bool ,
401 indent-broken-lines .default:n     = true ,
402 end-of-broken-line .tl_set:N        = \l_@@_end_of_broken_line_tl ,
403 end-of-broken-line .value_required:n = true ,
404 continuation-symbol .tl_set:N       = \l_@@_continuation_symbol_tl ,
405 continuation-symbol .value_required:n = true ,
406 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
407 continuation-symbol-on-indentation .value_required:n = true ,

```

```

408     unknown          .code:n =
409     \msg_error:nn { piton } { Unknown-key-for-PitonOptions }
410 }

```

The argument of \PitonOptions is provided by curryfication.

```

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

```

### 6.2.5 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` or `all-line-numbers`).

```

412 \int_new:N \g_@@_visual_line_int
413 \cs_new_protected:Npn \@@_print_number:
414 {
415     \int_gincr:N \g_@@_visual_line_int
416     \hbox_overlap_left:n
417     {
418         { \color { gray } \footnotesize \int_to_arabic:n \g_@@_visual_line_int }
419         \skip_horizontal:N \l_@@_numbers_sep_dim
420     }
421 }

```

### 6.2.6 The command to write on the aux file

```

422 \cs_new_protected:Npn \@@_write_aux:
423 {
424     \tl_if_empty:NF \g_@@_aux_tl
425     {
426         \iow_now:Nn \@mainaux { \ExplSyntaxOn }
427         \iow_now:Nx \@mainaux
428         {
429             \tl_gset:cn { c_@@_ \int_use:N \g_@@_env_int _ tl }
430             { \exp_not:V \g_@@_aux_tl }
431         }
432         \iow_now:Nn \@mainaux { \ExplSyntaxOff }
433     }
434     \tl_gclear:N \g_@@_aux_tl
435 }

```

The following macro will be used only when the key `width` is used with the special value `min`.

```

436 \cs_new_protected:Npn \@@_width_to_aux:
437 {
438     \tl_gput_right:Nx \g_@@_aux_tl
439     {
440         \dim_set:Nn \l_@@_line_width_dim
441         { \dim_eval:n { \g_@@_tmp_width_dim } }
442     }
443 }

```

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

```

444 \NewDocumentCommand { \piton } { }
445 { \peek_meaning:NTF \bgroup \@@_piton_standard \@@_piton_verbatim }
446 \NewDocumentCommand { \@@_piton_standard } { m }
447 {
448     \group_begin:

```

```
449 \ttfamily
```

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

```
450 \automatichyphenmode = 1
451 \cs_set_eq:NN \ \c_backslash_str
452 \cs_set_eq:NN \% \c_percent_str
453 \cs_set_eq:NN \{ \c_left_brace_str
454 \cs_set_eq:NN \} \c_right_brace_str
455 \cs_set_eq:NN \$ \c_dollar_str
456 \cs_set_eq:cN { ~ } \space
457 \cs_set_protected:Npn \@@_begin_line: { }
458 \cs_set_protected:Npn \@@_end_line: { }
459 \tl_set:Nx \l_tmpa_tl
460 {
461   \lua_now:e
462     { piton.ParseBis('\l_@@_language_str',token.scan_string()) }
463     { #1 }
464 }
465 \bool_if:NTF \l_@@_show_spaces_bool
466 { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
```

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.

```
467 {
468   \bool_if:NT \l_@@_break_lines_in_piton_bool
469   { \regex_replace_all:nnN { \x20 } { \x20 } \l_tmpa_tl }
470 }
471 \l_tmpa_tl
472 \group_end:
473 }
474 \NewDocumentCommand { \@@_piton_verbatim } { v }
475 {
476   \group_begin:
477   \ttfamily
478   \automatichyphenmode = 1
479   \cs_set_protected:Npn \@@_begin_line: { }
480   \cs_set_protected:Npn \@@_end_line: { }
481   \tl_set:Nx \l_tmpa_tl
482   {
483     \lua_now:e
484       { piton.Parse('\l_@@_language_str',token.scan_string()) }
485       { #1 }
486   }
487   \bool_if:NT \l_@@_show_spaces_bool
488   { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
489   \l_tmpa_tl
490   \group_end:
491 }
```

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).

```
492 \cs_new_protected:Npn \@@_piton:n #1
493 {
494   \group_begin:
495   \cs_set_protected:Npn \@@_begin_line: { }
496   \cs_set_protected:Npn \@@_end_line: { }
497   \bool_lazy_or:nnTF
498     \l_@@_break_lines_in_piton_bool
499     \l_@@_break_lines_in_Piton_bool
500   {
501     \tl_set:Nx \l_tmpa_tl
502     {
```



```

503         \lua_now:e
504         { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
505         { #1 }
506     }
507 }
508 {
509     \tl_set:Nx \l_tmpa_tl
510     {
511         \lua_now:e
512         { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
513         { #1 }
514     }
515 }
516 \bool_if:NT \l_@@_show_spaces_bool
517 { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
518 \l_tmpa_tl
519 \group_end:
520 }

```

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

```

521 \cs_new_protected:Npn \@@_piton_no_cr:n #1
522 {
523     \group_begin:
524     \cs_set_protected:Npn \@@_begin_line: { }
525     \cs_set_protected:Npn \@@_end_line: { }
526     \cs_set_protected:Npn \@@_newline:
527     { \msg_fatal:nn { piton } { cr~not~allowed } }
528     \bool_lazy_or:nnTF
529     \l_@@_break_lines_in_piton_bool
530     \l_@@_break_lines_in_Piton_bool
531     {
532         \tl_set:Nx \l_tmpa_tl
533         {
534             \lua_now:e
535             { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
536             { #1 }
537         }
538     }
539     {
540         \tl_set:Nx \l_tmpa_tl
541         {
542             \lua_now:e
543             { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
544             { #1 }
545         }
546     }
547     \bool_if:NT \l_@@_show_spaces_bool
548     { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
549     \l_tmpa_tl
550     \group_end:
551 }

```

Despite its name, \@@\_pre\_env: will be used both in \PitonInputFile and in the environments such as {Piton}.

```

552 \cs_new:Npn \@@_pre_env:
553 {
554     \automatichyphenmode = 1
555     \int_gincr:N \g_@@_env_int
556     \tl_gclear:N \g_@@_aux_tl
557     \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
558     { \dim_set_eq:NN \l_@@_width_dim \linewidth }

```

We read the information written on the aux file by previous run (when the key `width` is used with the special value `min`). At this time, the only potential information written on the aux file is the value of `\l_@@_line_width_dim` when the key `width` has been used with the special value `min`).

```

559 \cs_if_exist_use:c { c_@@ _ \int_use:N \g_@@_env_int _ t1 }
560 \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
561 \dim_gzero:N \g_@@_tmp_width_dim
562 \int_gzero:N \g_@@_line_int
563 \dim_zero:N \parindent
564 \dim_zero:N \lineskip
565 \dim_zero:N \parindent
566 \cs_set_eq:NN \label \@@_label:n
567 }

568 \keys_define:nn { PitonInputFile }
569 {
570   first-line .int_set:N = \l_@@_first_line_int ,
571   first-line .value_required:n = true ,
572   last-line .int_set:N = \l_@@_last_line_int ,
573   last-line .value_required:n = true ,
574 }

```

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

```

575 \cs_new_protected:Npn \@@_compute_left_margin:nn #1 #2
576 {
577   \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
578   {
579     \hbox_set:Nn \l_tmpa_box
580     {
581       \footnotesize
582       \bool_if:NTF \l_@@_all_line_numbers_bool
583       {
584         \int_to_arabic:n
585         { \g_@@_visual_line_int + \l_@@_nb_lines_int }
586       }
587       {
588         \lua_now:n
589         { piton.#1(token.scan_argument()) }
590         { #2 }
591         \int_to_arabic:n
592         { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
593       }
594     }
595     \dim_set:Nn \l_@@_left_margin_dim
596     { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
597   }
598 }

```

Whereas `\l_@@_with_dim` is the width of the environment, `\l_@@_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 `\l_@@_line_width_dim` from `\l_@@_width_dim` or we have to do the opposite.

```

599 \cs_new_protected:Npn \@@_compute_width:
600 {
601   \dim_compare:nNnTF \l_@@_line_width_dim = \c_zero_dim
602   {
603     \dim_set_eq:NN \l_@@_line_width_dim \l_@@_width_dim
604     \clist_if_empty:NTF \l_@@_bg_color_clist

```

If there is no background, we only subtract the left margin.

```

605         { \dim_sub:Nn \l_@@_line_width_dim \l_@@_left_margin_dim }

```

If there is a background, we subtract 0.5 em for the margin on the right.

```

606         {
607         \dim_sub:Nn \l_@@_line_width_dim { 0.5 em }

```

And we subtract also for the left margin. If the key `left-margin` has been used (with a numerical value or with the special value `min`), `\l_@@_left_margin_dim` has a non-zero value<sup>21</sup> and we use that value. Elsewhere, we use a value of 0.5 em.

```

608         \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
609         { \dim_sub:Nn \l_@@_line_width_dim { 0.5 em } }
610         { \dim_sub:Nn \l_@@_line_width_dim \l_@@_left_margin_dim }
611     }
612 }

```

If `\l_@@_line_width_dim` has yet a non-empty value, that means that it has been read on the `aux` file: it has been written on a previous run because the key `width` is used with the special value `min`). We compute now the width of the environment by computations opposite to the preceding ones.

```

613 {
614     \dim_set_eq:Nn \l_@@_width_dim \l_@@_line_width_dim
615     \clist_if_empty:NTF \l_@@_bg_color_clist
616     { \dim_add:Nn \l_@@_width_dim \l_@@_left_margin_dim }
617     {
618         \dim_add:Nn \l_@@_width_dim { 0.5 em }
619         \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
620         { \dim_add:Nn \l_@@_width_dim { 0.5 em } }
621         { \dim_add:Nn \l_@@_width_dim \l_@@_left_margin_dim }
622     }
623 }
624 }

```

```

625 \NewDocumentCommand { \NewPitonEnvironment } { m m m m }
626 {

```

We construct a TeX macro which will catch as argument all the tokens until `\end{name_env}` with, in that `\end{name_env}`, the catcodes of `\`, `{` and `}` equal to 12 (“other”). The latter explains why the definition of that function is a bit complicated.

```

627     \use:x
628     {
629         \cs_set_protected:Npn
630         \use:c { _@@_collect_ #1 :w }
631         ####1
632         \c_backslash_str end \c_left_brace_str #1 \c_right_brace_str
633     }
634     {
635         \group_end:
636         \mode_if_vertical:TF \mode_leave_vertical: \newline

```

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.

```

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

```

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.

```

638         @@_compute_left_margin:nn { CountNonEmptyLines } { ##1 }
639         @@_compute_width:
640         \ttfamily
641         \dim_zero:N \parskip % added 2023/07/06

```

`\c_@@_footnote_bool` is raised when the package `piton` has been load with the key `footnote` or the key `footnotehyper`.

```

642         \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
643         \vtop \bgroup

```

---

<sup>21</sup>If the key `left-margin` has been used with the special value `min`, the actual value of `\l_@@_left_margin_dim` has yet been computed when we use the current command.

```

644 \lua_now:e
645 {
646     piton.GobbleParse
647     (
648         '\l_@@_language_str' ,
649         \int_use:N \l_@@_gobble_int ,
650         token.scan_argument()
651     )
652 }
653 { ##1 }
654 \vspace { 2.5 pt }
655 \egroup
656 \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }

```

If the user has used the key `width` with the special value `min`, we write on the `aux` file the value of `\l_@@_line_width_dim` (largest width of the lines of code of the environment).

```

657 \bool_if:NT \l_@@_width_min_bool \@@_width_to_aux:

```

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

```

658 \end { #1 }
659 \@@_write_aux:
660 }

```

We can now define the new environment.

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

```

661 \NewDocumentEnvironment { #1 } { #2 }
662 {
663     #3
664     \@@_pre_env:
665     \group_begin:
666     \tl_map_function:nN
667     { \ \ \ \{ \} \$ \% \# \^ \_ \% \~ \^^I }
668     \char_set_catcode_other:N
669     \use:c { _@@_collect_ #1 :w }
670 }
671 { #4 }

```

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

```

672 \AddToHook { env / #1 / begin } { \char_set_catcode_other:N ^^M }
673 }

```

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

Now, we define the environment `{Piton}`, which is the main environment provided by the package `piton`. Of course, you use `\NewPitonEnvironment`.

```

674 \bool_if:NTF \c_@@_beamer_bool
675 {
676     \NewPitonEnvironment { Piton } { d < > }
677     {
678         \IfValueTF { #1 }
679         { \begin { uncoverenv } < #1 > }
680         { \begin { uncoverenv } }
681     }
682     { \end { uncoverenv } }
683 }
684 { \NewPitonEnvironment { Piton } { } { } { } { } }

```

The code of the command `\PitonInputFile` is somewhat similar to the code of the environment `{Piton}`. In fact, it's simpler because there isn't the problem of catching the content of the environment in a verbatim mode.

```

685 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
686 {
687   \file_if_exist:nTF { #3 }
688     { \@@_input_file:nnn { #1 } { #2 } { #3 } }
689     { \msg_error:nnn { piton } { unknown-file } { #3 } }
690 }
691 \cs_new_protected:Npn \@@_input_file:nnn #1 #2 #3
692 {

```

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

```

693   \tl_if_novalue:nF { #1 }
694   {
695     \bool_if:NTF \c_@@_beamer_bool
696       { \begin { uncoverenv } < #1 > }
697       { \msg_error:nn { piton } { overlay~without~beamer } }
698   }
699   \group_begin:
700     \int_zero_new:N \l_@@_first_line_int
701     \int_zero_new:N \l_@@_last_line_int
702     \int_set_eq:NN \l_@@_last_line_int \c_max_int
703     \keys_set:nn { PitonInputFile } { #2 }
704     \@@_pre_env:
705     \mode_if_vertical:TF \mode_leave_vertical: \newline

```

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.

```

706   \lua_now:n { piton.CountLinesFile(token.scan_argument()) } { #3 }

```

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.

```

707   \@@_compute_left_margin:nn { CountNonEmptyLinesFile } { #3 }
708   \@@_compute_width:
709   \ttfamily
710   \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
711   \vtop \bgroup
712   \lua_now:e
713   {
714     piton.ParseFile(
715       '\l_@@_language_str',
716       token.scan_argument() ,
717       \int_use:N \l_@@_first_line_int ,
718       \int_use:N \l_@@_last_line_int )
719   }
720   { #3 }
721   \egroup
722   \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
723   \bool_if:NT \l_@@_width_min_bool \@@_width_to_aux:
724   \group_end:

```

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.

```

725   \tl_if_novalue:nF { #1 }
726   { \bool_if:NT \c_@@_beamer_bool { \end { uncoverenv } } }
727   \@@_write_aux:
728 }

```

### 6.2.8 The styles

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

```

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

```

The following command takes in its argument by curryfication.

```

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

```

```

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

733 \keys_define:nn { piton / Styles }
734 {
735   String.Interpol .tl_set:c = pitonStyle String.Interpol ,
736   String.Interpol .value_required:n = true ,
737   FormattingType .tl_set:c = pitonStyle FormattingType ,
738   FormattingType .value_required:n = true ,
739   Dict.Value .tl_set:c = pitonStyle Dict.Value ,
740   Dict.Value .value_required:n = true ,
741   Name.Decorator .tl_set:c = pitonStyle Name.Decorator ,
742   Name.Decorator .value_required:n = true ,
743   Name.Field .tl_set:c = pitonStyle Name.Field ,
744   Name.Field .value_required:n = true ,
745   Name.Function .tl_set:c = pitonStyle Name.Function ,
746   Name.Function .value_required:n = true ,
747   UserFunction .tl_set:c = pitonStyle UserFunction ,
748   UserFunction .value_required:n = true ,
749   Keyword .tl_set:c = pitonStyle Keyword ,
750   Keyword .value_required:n = true ,
751   Keyword.Constant .tl_set:c = pitonStyle Keyword.Constant ,
752   Keyword.constant .value_required:n = true ,
753   String.Doc .tl_set:c = pitonStyle String.Doc ,
754   String.Doc .value_required:n = true ,
755   Interpol.Inside .tl_set:c = pitonStyle Interpol.Inside ,
756   Interpol.Inside .value_required:n = true ,
757   String.Long .tl_set:c = pitonStyle String.Long ,
758   String.Long .value_required:n = true ,
759   String.Short .tl_set:c = pitonStyle String.Short ,
760   String.Short .value_required:n = true ,
761   String .meta:n = { String.Long = #1 , String.Short = #1 } ,
762   Comment.Math .tl_set:c = pitonStyle Comment.Math ,
763   Comment.Math .default:n = \@@_math_scantokens:n ,
764   Comment.Math .initial:n = ,
765   Comment .tl_set:c = pitonStyle Comment ,
766   Comment .value_required:n = true ,
767   Name.Constructor .tl_set:c = pitonStyle Name.Constructor ,
768   Name.Constructor .value_required:n = true ,
769   InitialValues .tl_set:c = pitonStyle InitialValues ,
770   InitialValues .value_required:n = true ,
771   Number .tl_set:c = pitonStyle Number ,
772   Number .value_required:n = true ,
773   Name.Namespace .tl_set:c = pitonStyle Name.Namespace ,
774   Name.Namespace .value_required:n = true ,
775   Name.Module .tl_set:c = pitonStyle Name.Module ,
776   Name.Module .value_required:n = true ,
777   Name.Class .tl_set:c = pitonStyle Name.Class ,
778   Name.Class .value_required:n = true ,
779   Name.Builtin .tl_set:c = pitonStyle Name.Builtin ,
780   Name.Builtin .value_required:n = true ,
781   TypeParameter .tl_set:c = pitonStyle TypeParameter ,
782   TypeParameter .value_required:n = true ,
783   Name.Type .tl_set:c = pitonStyle Name.Type ,
784   Name.Type .value_required:n = true ,
785   Operator .tl_set:c = pitonStyle Operator ,
786   Operator .value_required:n = true ,
787   Operator.Word .tl_set:c = pitonStyle Operator.Word ,
788   Operator.Word .value_required:n = true ,
789   Exception .tl_set:c = pitonStyle Exception ,
790   Exception .value_required:n = true ,
791   Comment.LaTeX .tl_set:c = pitonStyle Comment.LaTeX ,
792   Comment.LaTeX .value_required:n = true ,

```

```

793 Identifier      .tl_set:c = pitonStyle Identifier ,
794 Comment.LaTeX    .value_required:n = true ,
795 ParseAgain.noCR  .tl_set:c = pitonStyle ParseAgain.noCR ,
796 ParseAgain.noCR  .value_required:n = true ,
797 ParseAgain       .tl_set:c = pitonStyle ParseAgain ,
798 ParseAgain       .value_required:n = true ,
799 Prompt          .tl_set:c = pitonStyle Prompt ,
800 Prompt          .value_required:n = true ,
801 unknown         .code:n =
802   \msg_error:nn { piton } { Unknown~key~for~SetPitonStyle }
803 }

804 \msg_new:nnn { piton } { Unknown~key~for~SetPitonStyle }
805 {
806   The~style~'\l_keys_key_str'~is~unknown.\\
807   This~key~will~be~ignored.\\
808   The~available~styles~are~(in~alphabetic~order):~
809   Comment,~
810   Comment.LaTeX,~
811   Dict.Value,~
812   Exception,~
813   Identifier,~
814   InitialValues,~
815   Keyword,~
816   Keyword.Constant,~
817   Name.Builtin,~
818   Name.Class,~
819   Name.Constructor,~
820   Name.Decorator,~
821   Name.Field,~
822   Name.Function,~
823   Name.Module,~
824   Name.Namespace,~
825   UserFunction,~
826   Number,~
827   Operator,~
828   Operator.Word,~
829   Prompt,~
830   String,~
831   String.Doc,~
832   String.Long,~
833   String.Short,~and~
834   String.Interpol.
835 }

```

### 6.2.9 The initial styles

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

```

836 \SetPitonStyle
837 {
838   Comment      = \color[HTML]{0099FF} \itshape ,
839   Exception     = \color[HTML]{CC0000} ,
840   Keyword      = \color[HTML]{006699} \bfseries ,
841   Keyword.Constant = \color[HTML]{006699} \bfseries ,
842   Name.Builtin  = \color[HTML]{336666} ,
843   Name.Decorator = \color[HTML]{9999FF} ,
844   Name.Class    = \color[HTML]{00AA88} \bfseries ,
845   Name.Function = \color[HTML]{CC00FF} ,
846   Name.Namespace = \color[HTML]{00CCFF} ,
847   Name.Constructor = \color[HTML]{006000} \bfseries ,

```

```

848 Name.Field          = \color[HTML]{AA6600} ,
849 Name.Module         = \color[HTML]{0060A0} \bfseries ,
850 Number              = \color[HTML]{FF6600} ,
851 Operator            = \color[HTML]{555555} ,
852 Operator.Word       = \bfseries ,
853 String              = \color[HTML]{CC3300} ,
854 String.Doc          = \color[HTML]{CC3300} \itshape ,
855 String.Interpol      = \color[HTML]{AA0000} ,
856 Comment.LaTeX       = \normalfont \color[rgb]{.468,.532,.6} ,
857 Name.Type           = \color[HTML]{336666} ,
858 InitialValues       = \@@_piton:n ,
859 Dict.Value          = \@@_piton:n ,
860 Interpol.Inside     = \color{black}\@@_piton:n ,
861 TypeParameter       = \color[HTML]{336666} \itshape ,
862 Identifier          = \@@_identifier:n ,
863 UserFunction = ,
864 Prompt              = ,
865 ParseAgain.noCR     = \@@_piton_no_cr:n ,
866 ParseAgain          = \@@_piton:n ,
867 }

```

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).

```

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

```

### 6.2.10 Highlighting some identifiers

```

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

871 \keys_define:nn { PitonOptions }
872 { identifiers .code:n = \@@_set_identifiers:n { #1 } }

873 \keys_define:nn { Piton / identifiers }
874 {
875   names .clist_set:N = \l_@@_identifiers_names_tl ,
876   style .tl_set:N    = \l_@@_style_tl ,
877 }

878 \cs_new_protected:Npn \@@_set_identifiers:n #1
879 {
880   \clist_clear_new:N \l_@@_identifiers_names_tl
881   \tl_clear_new:N \l_@@_style_tl
882   \keys_set:nn { Piton / identifiers } { #1 }
883   \clist_map_inline:Nn \l_@@_identifiers_names_tl
884   {
885     \tl_set_eq:cN
886     { PitonIdentifier _ \l_@@_language_str _ ##1 }
887     \l_@@_style_tl
888   }
889 }

```

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`).

```
890 \cs_new_protected:cpn { pitonStyle Name.Function.Internal } #1
891 {
```

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

```
892 { \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 affectation is global because we have to exit many groups and even the environments `{Piton}`.

```
893 \cs_gset_protected:cpn { PitonIdentifier _ \l_@@_language_str _ #1 }
894 { \PitonStyle { UserFunction } }
```

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`.**

```
895 \seq_if_exist:cF { g_@@_functions _ \l_@@_language_str _ seq }
896 { \seq_new:c { g_@@_functions _ \l_@@_language_str _ seq } }
897 \seq_gput_right:cn { g_@@_functions _ \l_@@_language_str _ seq } { #1 }
898 }
```

```
899 \NewDocumentCommand \PitonClearUserFunctions { ! 0 { \l_@@_language_str } }
900 {
901   \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
902   {
903     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
904     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }
905     \seq_gclear:c { g_@@_functions _ #1 _ seq }
906   }
907 }
```

### 6.2.11 Security

```
908 \AddToHook { env / piton / begin }
909 { \msg_fatal:nn { piton } { No-environment-piton } }
910
911 \msg_new:nnn { piton } { No-environment-piton }
912 {
913   There-is-no-environment-piton!\
914   There-is-an-environment~{Piton}~and~a~command~
915   \token_to_str:N \piton\ but~there-is-no-environment~
916   {piton}~.~This~error~is~fatal.
917 }
```

### 6.2.12 The error messages of the package

```
918 \msg_new:nnn { piton } { syntax-error }
919 {
920   Your~code~is~not~syntactically~correct.\
921   It~won't~be~printed~in~the~PDF~file.
922 }
923 \NewDocumentCommand \PitonSyntaxError { }
924 { \msg_error:nn { piton } { syntax-error } }
925 \msg_new:nnn { piton } { unknown-file }
926 {
927   Unknown~file. \
928   The~file~'#1'~is~unknown.\
929   Your~command~\token_to_str:N \PitonInputFile\ will~be~discarded.
930 }
931 \msg_new:nnnn { piton } { Unknown-key-for-PitonOptions }
932 {
933   Unknown~key. \
```

```

934 The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
935 It~will~be~ignored.\\
936 For~a~list~of~the~available~keys,~type~H~<return>.
937 }
938 {
939 The~available~keys~are~(in~alphabetic~order):~
940 all~line~numbers,~
941 auto~gobble,~
942 background~color,~
943 break~lines,~
944 break~lines~in~piton,~
945 break~lines~in~Piton,~
946 continuation~symbol,~
947 continuation~symbol~on~indentation,~
948 end~of~broken~line,~
949 env~gobble,~
950 gobble,~
951 identifiers,~
952 indent~broken~lines,~
953 language,~
954 left~margin,~
955 line~numbers,~
956 prompt~background~color,~
957 resume,~
958 show~spaces,~
959 show~spaces~in~strings,~
960 splittable,~
961 tabs~auto~gobble,~
962 tab~size~and~width.
963 }

964 \msg_new:nnn { piton } { label~with~lines~numbers }
965 {
966 You~can't~use~the~command~\token_to_str:N \label\
967 because~the~key~'line~numbers'~(or~'all~line~numbers')~
968 is~not~active.\\
969 If~you~go~on,~that~command~will~ignored.
970 }

971 \msg_new:nnn { piton } { cr~not~allowed }
972 {
973 You~can't~put~any~carriage~return~in~the~argument~
974 of~a~command~\c_backslash_str
975 \l_@@_beamer_command_str\ within~an~
976 environment~of~'piton'.~You~should~consider~using~the~
977 corresponding~environment.\\
978 That~error~is~fatal.
979 }

980 \msg_new:nnn { piton } { overlay~without~beamer }
981 {
982 You~can't~use~an~argument~<...>~for~your~command~
983 \token_to_str:N \PitonInputFile\ because~you~are~not~
984 in~Beamer.\\
985 If~you~go~on,~that~argument~will~be~ignored.
986 }

987 \msg_new:nnn { Piton } { Python~error }
988 { A~Python~error~has~been~detected. }

```

## 6.3 The Lua part of the implementation

```

989 \ExplSyntaxOff
990 \RequirePackage{luacode}

```

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`.

```

991 \begin{luacode*}
992 piton = piton or { }
993 if piton.comment_latex == nil then piton.comment_latex = ">" end
994 piton.comment_latex = "#" .. piton.comment_latex

```

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

```

995 function piton.open_brace ()
996     tex.sprint("{")
997 end
998 function piton.close_brace ()
999     tex.sprint("}")
1000 end

```

### 6.3.1 Special functions dealing with LPEG

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

```

1001 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
1002 local Cf, Cs, Cg, Cmt, Cb = lpeg.Cf, lpeg.Cs, lpeg.Cg, lpeg.Cmt, lpeg.Cb
1003 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”).

```

1004 local function Q(pattern)
1005     return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
1006 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 “`escape-inside`”. That function won't be much used.

```

1007 local function L(pattern)
1008     return Ct ( C ( pattern ) )
1009 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.

```

1010 local function Lc(string)
1011     return Cc ( { luatexbase.catcodetables.expl, string } )
1012 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)

```

1013 local function K(style, pattern)
1014     return

```

```

1015     Lc ( "{\\PitonStyle{" .. style .. "}{" )
1016     * Q ( pattern )
1017     * Lc ( "}}" )
1018 end

```

The formatting commands in a given `piton` style (eg. the style `Keyword`) may be semi-global declarations (such as `\bfseries` or `\slshape`) 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}}`.

```

1019 local function WithStyle(style,pattern)
1020     return
1021         Ct ( Cc "Open" * Cc ( "{\\PitonStyle{" .. style .. "}{" ) * Cc "}}" )
1022     * pattern
1023     * Ct ( Cc "Close" )
1024 end

```

The following LPEG catches the Python chunks which are in LaTeX escapes (and that chunks will be considered as normal LaTeX constructions). We recall that `piton.begin_espace` and `piton.end_escape` are Lua strings corresponding to the key `escape-inside`<sup>22</sup>. Since the elements that will be caught 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`) without number of catcode table at the first component of the table.

```

1025 local Escape =
1026     P(piton_begin_escape)
1027     * L ( ( 1 - P(piton_end_escape) ) ^ 1 )
1028     * P(piton_end_escape)

```

The following line is mandatory.

```

1029 lpeg.locale(lpeg)

```

### The basic syntactic LPEG

```

1030 local alpha, digit = lpeg.alpha, lpeg.digit
1031 local space = P " "

```

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

```

1032 local letter = alpha + P "_"
1033     + P "â" + P "à" + P "ç" + P "é" + P "è" + P "ê" + P "ë" + P "ï" + P "î"
1034     + P "ô" + P "û" + P "ü" + P "Â" + P "Ã" + P "Ç" + P "É" + P "È" + P "Ê"
1035     + P "Ë" + P "Ï" + P "Î" + P "Ï" + P "Ô" + P "Õ" + P "Ü"
1036
1037 local alphanum = letter + digit

```

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

```

1038 local identifier = letter * alphanum ^ 0

```

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

```

1039 local Identifier = K ( 'Identifier' , identifier)

```

---

<sup>22</sup>The `piton` key `escape-inside` is available at load-time only.

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

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

```

1040 local Number =
1041   K ( 'Number' ,
1042       ( digit^1 * P "." * digit^0 + digit^0 * P "." * digit^1 + digit^1 )
1043       * ( S "eE" * S "+-" ^ -1 * digit^1 ) ^ -1
1044       + digit^1
1045   )

```

We recall that `piton.begin_escape` and `piton.end_escape` are Lua strings corresponding to the key `escape-inside`<sup>23</sup>. Of course, if the final user has not used the key `escape-inside`, these strings are empty.

```

1046 local Word
1047 if piton_begin_escape ~= ''
1048 then Word = Q ( ( ( 1 - space - P(piton_begin_escape) - P(piton_end_escape) )
1049                 - S "'\"\\r[()]" - digit ) ^ 1 )
1050 else Word = Q ( ( ( 1 - space ) - S "'\"\\r[()]" - digit ) ^ 1 )
1051 end

1052 local Space = ( Q " " ) ^ 1
1053
1054 local SkipSpace = ( Q " " ) ^ 0
1055
1056 local Punct = Q ( S ".,:;!" )
1057
1058 local Tab = P "\t" * Lc ( '\\l_@@_tab_t1' )

1059 local SpaceIndentation = Lc ( '\\l_@@_an_indentation_space:' ) * ( Q " " )

1060 local Delim = Q ( S "[()]" )

```

The following LPEG catches a space (U+0020) and replace it by `\\l_@@_space_t1`. It will be used in the strings. Usually, `\\l_@@_space_t1` will contain a space and therefore there won't be difference. However, when the key `show-spaces-in-strings` is in force, `\\l_@@_space_t1` will contain `□` (U+2423) in order to visualize the spaces.

```

1061 local VisualSpace = space * Lc "\\l_@@_space_t1"

```

### 6.3.2 The LPEG python

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

```

1062 local Operator =
1063   K ( 'Operator' ,
1064       P "!=" + P "<>" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P "!="
1065       + P "/" + P "*" + S "-~+/*%=<>&.@|"
1066   )
1067
1068 local OperatorWord =

```

<sup>23</sup>The `piton` key `escape-inside` is available at load-time only.

```

1069 K ( 'Operator.Word' , P "in" + P "is" + P "and" + P "or" + P "not" )
1070
1071 local Keyword =
1072 K ( 'Keyword' ,
1073     P "as" + P "assert" + P "break" + P "case" + P "class" + P "continue"
1074     + P "def" + P "del" + P "elif" + P "else" + P "except" + P "exec"
1075     + P "finally" + P "for" + P "from" + P "global" + P "if" + P "import"
1076     + P "lambda" + P "non local" + P "pass" + P "return" + P "try"
1077     + P "while" + P "with" + P "yield" + P "yield from" )
1078 + K ( 'Keyword.Constant' , P "True" + P "False" + P "None" )
1079
1080 local Builtin =
1081 K ( 'Name.Builtin' ,
1082     P "__import__" + P "abs" + P "all" + P "any" + P "bin" + P "bool"
1083     + P "bytearray" + P "bytes" + P "chr" + P "classmethod" + P "compile"
1084     + P "complex" + P "delattr" + P "dict" + P "dir" + P "divmod"
1085     + P "enumerate" + P "eval" + P "filter" + P "float" + P "format"
1086     + P "frozenset" + P "getattr" + P "globals" + P "hasattr" + P "hash"
1087     + P "hex" + P "id" + P "input" + P "int" + P "isinstance" + P "issubclass"
1088     + P "iter" + P "len" + P "list" + P "locals" + P "map" + P "max"
1089     + P "memoryview" + P "min" + P "next" + P "object" + P "oct" + P "open"
1090     + P "ord" + P "pow" + P "print" + P "property" + P "range" + P "repr"
1091     + P "reversed" + P "round" + P "set" + P "setattr" + P "slice" + P "sorted"
1092     + P "staticmethod" + P "str" + P "sum" + P "super" + P "tuple" + P "type"
1093     + P "vars" + P "zip" )
1094
1095
1096 local Exception =
1097 K ( 'Exception' ,
1098     P "ArithmeticError" + P "AssertionError" + P "AttributeError"
1099     + P "BaseException" + P "BufferError" + P "BytesWarning" + P "DeprecationWarning"
1100     + P "EOFError" + P "EnvironmentError" + P "Exception" + P "FloatingPointError"
1101     + P "FutureWarning" + P "GeneratorExit" + P "IOError" + P "ImportError"
1102     + P "ImportWarning" + P "IndentationError" + P "IndexError" + P "KeyError"
1103     + P "KeyboardInterrupt" + P "LookupError" + P "MemoryError" + P "NameError"
1104     + P "NotImplementedError" + P "OSError" + P "OverflowError"
1105     + P "PendingDeprecationWarning" + P "ReferenceError" + P "ResourceWarning"
1106     + P "RuntimeError" + P "RuntimeWarning" + P "StopIteration"
1107     + P "SyntaxError" + P "SyntaxWarning" + P "SystemError" + P "SystemExit"
1108     + P "TabError" + P "TypeError" + P "UnboundLocalError" + P "UnicodeDecodeError"
1109     + P "UnicodeEncodeError" + P "UnicodeError" + P "UnicodeTranslateError"
1110     + P "UnicodeWarning" + P "UserWarning" + P "ValueError" + P "VMSError"
1111     + P "Warning" + P "WindowsError" + P "ZeroDivisionError"
1112     + P "BlockingIOError" + P "ChildProcessError" + P "ConnectionError"
1113     + P "BrokenPipeError" + P "ConnectionAbortedError" + P "ConnectionRefusedError"
1114     + P "ConnectionResetError" + P "FileExistsError" + P "FileNotFoundError"
1115     + P "InterruptedError" + P "IsADirectoryError" + P "NotADirectoryError"
1116     + P "PermissionError" + P "ProcessLookupError" + P "TimeoutError"
1117     + P "StopAsyncIteration" + P "ModuleNotFoundError" + P "RecursionError" )
1118
1119
1120 local RaiseException = K ( 'Keyword' , P "raise" ) * SkipSpace * Exception * Q ( P "(" )
1121

```

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

```

1122 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:`

```

1123 local DefClass =

```

```
1124 K ( 'Keyword' , P "class" ) * Space * K ( 'Name.Class' , identifier )
```

If the word `class` is not followed by an identifier, it will be caught 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 `python` 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`

```
1125 local ImportAs =
1126   K ( 'Keyword' , P "import" )
1127   * Space
1128   * K ( 'Name.Namespace' ,
1129       identifier * ( P "." * identifier ) ^ 0 )
1130   * (
1131       ( Space * K ( 'Keyword' , P "as" ) * Space
1132         * K ( 'Name.Namespace' , identifier ) )
1133       +
1134       ( SkipSpace * Q ( P "," ) * SkipSpace
1135         * K ( 'Name.Namespace' , identifier ) ) ^ 0
1136   )
```

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 `python` style `Name.Namespace` and the following keyword `import` must be formatted with the `python` style `Keyword` and must *not* be caught by the LPEG `ImportAs`.

Example: `from math import pi`

```
1137 local FromImport =
1138   K ( 'Keyword' , P "from" )
1139   * Space * K ( 'Name.Namespace' , identifier )
1140   * 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.

	Single	Double
Short	'text'	"text"
Long	'''test'''	"""test"""

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<sup>24</sup> in that interpolation:

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

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

```
1141 local PercentInterpol =
1142   K ( 'String.Interpol' ,
1143       P "%"
```

<sup>24</sup>There is no special `python` 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`.

```

1144 * ( P "(" * alphanum ^ 1 * P ")" ) ^ -1
1145 * ( S "-#0 +" ) ^ 0
1146 * ( digit ^ 1 + P "*" ) ^ -1
1147 * ( P "." * ( digit ^ 1 + P "*" ) ) ^ -1
1148 * ( S "HLL" ) ^ -1
1149 * S "sdfFeExXorgiGauc%"
1150 )

```

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.<sup>25</sup>

```

1151 local SingleShortString =
1152   WithStyle ( 'String.Short' ,

```

First, we deal with the f-strings of Python, which are prefixed by `f` or `F`.

```

1153   Q ( P "f'" + P "F'" )
1154   * (
1155     K ( 'String.Interpol' , P "{" )
1156     * K ( 'Interpol.Inside' , ( 1 - S "':" ) ^ 0 )
1157     * Q ( P ":" * ( 1 - S "':" ) ^ 0 ) ^ -1
1158     * K ( 'String.Interpol' , P "}" )
1159     +
1160     VisualSpace
1161     +
1162     Q ( ( P "\\'" + P "{" + P "}" + 1 - S "{'" ) ^ 1 )
1163     ) ^ 0
1164   * Q ( P "'" )
1165   +

```

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

```

1166   Q ( P '"' + P "r'" + P "R'" )
1167   * ( Q ( ( P "\\'" + 1 - S " 'r'" ) ^ 1 )
1168     + VisualSpace
1169     + PercentInterpol
1170     + Q ( P "%" )
1171     ) ^ 0
1172   * Q ( P '"' )
1173
1174
1175 local DoubleShortString =
1176   WithStyle ( 'String.Short' ,
1177     Q ( P "f\\" + P "F\\" )
1178     * (
1179       K ( 'String.Interpol' , P "{" )
1180       * Q ( ( 1 - S "':" ) ^ 0 , 'Interpol.Inside' )
1181       * ( K ( 'String.Interpol' , P ":" ) * Q ( ( 1 - S "':" ) ^ 0 ) ) ^ -1
1182       * K ( 'String.Interpol' , P "}" )
1183       +
1184       VisualSpace
1185       +
1186       Q ( ( P "\\\\" + P "{" + P "}" + 1 - S "{\\" ) ^ 1 )
1187       ) ^ 0
1188     * Q ( P "\\" )
1189     +
1190     Q ( P "\\\\" + P "r\\" + P "R\\" )
1191     * ( Q ( ( P "\\\\" + 1 - S " '\\r'" ) ^ 1 )
1192       + VisualSpace
1193       + PercentInterpol
1194       + Q ( P "%" )
1195       ) ^ 0

```

<sup>25</sup>The interpolations are formatted with the `piton` style `Interpol.Inside`. The initial value of that style is `\\@@_piton:n` wich means that the interpolations are parsed once again by `piton`.



```

1196         * Q ( P "\" ) )
1197
1198 local ShortString = SingleShortString + DoubleShortString

```

**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).

```

1199 local balanced_braces =
1200   P { "E" ,
1201       E =
1202         (
1203           P "{" * V "E" * P "}"
1204         +
1205           ShortString
1206         +
1207           ( 1 - S "{}" )
1208         ) ^ 0
1209   }

```

If Beamer is used (or if the key `beamer` is used at load-time), the following LPEG will be redefined.

```

1210 local Beamer = P ( false )
1211 local BeamerBeginEnvironments = P ( true )
1212 local BeamerEndEnvironments = P ( true )
1213 local BeamerNamesEnvironments =
1214   P "uncoverenv" + P "onlyenv" + P "visibleenv" + P "invisibleenv"
1215   + P "alertenv" + P "actionenv"

```

The following function will return a LPEG which will catch an environment of Beamer (supported by `piton`), that is to say `{uncover}`, `{only}`, etc.

```

1216 function OneBeamerEnvironment(name)
1217   return
1218     Ct ( Cc "Open"
1219         * C (
1220           P ( "\\begin{" .. name .. "}" )
1221           * ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1
1222         )
1223         * Cc ( "\\end{" .. name .. "}" )
1224       )
1225     * (
1226       C ( ( 1 - P ( "\\end{" .. name .. "}" ) ) ^ 0 )
1227       / ( function (s) return MainLoopPython:match(s) end )
1228     )
1229     * P ( "\\end{" .. name .. "}" ) * Ct ( Cc "Close" )
1230   end

1231 if piton_beamer
1232 then
1233   Beamer =
1234     L ( P "\\pause" * ( P "[" * ( 1 - P "]" ) ^ 0 * P "]" ) ^ -1 )
1235     +
1236     Ct ( Cc "Open"
1237         * C (
1238           (
1239             P "\\uncover" + P "\\only" + P "\\alert" + P "\\visible"
1240             + P "\\invisible" + P "\\action"
1241           )
1242           * ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1
1243           * P "{"

```

```

1244         )
1245         * Cc "}"
1246     )
1247     * ( C ( balanced_braces ) / (function (s) return MainLoopPython:match(s) end ) )
1248     * P "}" * Ct ( Cc "Close" )
1249     + OneBeamerEnvironment "uncoverenv"
1250     + OneBeamerEnvironment "onlyenv"
1251     + OneBeamerEnvironment "visibleenv"
1252     + OneBeamerEnvironment "invisibleenv"
1253     + OneBeamerEnvironment "alertenv"
1254     + OneBeamerEnvironment "actionenv"
1255     +
1256     L (

```

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

```

1257     ( P "\\alt" )
1258     * P "<" * ( 1 - P ">" ) ^ 0 * P ">"
1259     * P "{"
1260 )
1261 * K ( 'ParseAgain.noCR' , balanced_braces )
1262 * L ( P "}" )
1263 * K ( 'ParseAgain.noCR' , balanced_braces )
1264 * L ( P "}" )
1265 +
1266 L (

```

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

```

1267     ( P "\\temporal" )
1268     * P "<" * ( 1 - P ">" ) ^ 0 * P ">"
1269     * P "{"
1270 )
1271 * K ( 'ParseAgain.noCR' , balanced_braces )
1272 * L ( P "}" )
1273 * K ( 'ParseAgain.noCR' , balanced_braces )
1274 * L ( P "}" )
1275 * K ( 'ParseAgain.noCR' , balanced_braces )
1276 * L ( P "}" )

```

Now for the environemnts.

```

1277 BeamerBeginEnvironments =
1278     ( space ^ 0 *
1279         L
1280         (
1281             P "\\begin{" * BeamerNamesEnvironments * "}"
1282             * ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1
1283         )
1284         * P "\r"
1285     ) ^ 0
1286 BeamerEndEnvironments =
1287     ( space ^ 0 *
1288         L ( P "\\end{" * BeamerNamesEnvironments * P "}" )
1289         * P "\r"
1290     ) ^ 0
1291 end

```

**EOL** The following LPEG will detect the Python prompts when the user is typesetting an interactive session of Python (directly or through `{pyconsole}` of `pyluatex`). 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 de background).

```

1292 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).

```
1293 local Prompt = K ( 'Prompt' , ( ( P ">>>" + P "..." ) * P " " ^ -1 ) ^ -1 )
```

The following LPEG EOL is for the end of lines.

```
1294 local EOL =
1295   P "\r"
1296   *
1297   (
1298     ( space^0 * -1 )
1299     +
```

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:`<sup>26</sup>.

```
1300   Ct (
1301     Cc "EOL"
1302     *
1303     Ct (
1304       Lc "\\@@_end_line:"
1305       * BeamerEndEnvironments
1306       * BeamerBeginEnvironments
1307       * PromptHastyDetection
1308       * Lc "\\@@_newline: \\@@_begin_line:"
1309       * Prompt
1310     )
1311   )
1312 )
1313 *
1314 SpaceIndentation ^ 0
```

## The long strings

```
1315 local SingleLongString =
1316   WithStyle ( 'String.Long' ,
1317     ( Q ( S "fF" * P "'''" )
1318       * (
1319         K ( 'String.Interpol' , P "{" )
1320         * K ( 'Interpol.Outside' , ( 1 - S "};\r" - P "'''" ) ^ 0 )
1321         * Q ( P ":" * ( 1 - S "};\r" - P "'''" ) ^ 0 ) ^ -1
1322         * K ( 'String.Interpol' , P "}" )
1323         +
1324         Q ( ( 1 - P "'''" - S "{"'\r" ) ^ 1 )
1325         +
1326         EOL
1327       ) ^ 0
1328     +
1329     Q ( ( S "rR" ) ^ -1 * P "'''" )
1330     * (
1331       Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
1332       +
1333       PercentInterpol
1334       +
1335       P "%"
1336       +
1337       EOL
```

---

<sup>26</sup>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:`

```

1338         ) ^ 0
1339     )
1340     * Q ( P "'''" ) )
1341
1342
1343 local DoubleLongString =
1344     WithStyle ( 'String.Long' ,
1345         (
1346             Q ( S "fF" * P "\"\\\"" )
1347             * (
1348                 K ( 'String.Interpol', P "{" )
1349                 * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - P "\"\\\"" ) ^ 0 )
1350                 * Q ( P ":" * ( 1 - S "}:\\r" - P "\"\\\"" ) ^ 0 ) ^ -1
1351                 * K ( 'String.Interpol' , P "}" )
1352                 +
1353                 Q ( ( 1 - P "\"\\\"" - S "{\\r" ) ^ 1 )
1354                 +
1355                 EOL
1356             ) ^ 0
1357         +
1358         Q ( ( S "rR" ) ^ -1 * P "\"\\\"" )
1359         * (
1360             Q ( ( 1 - P "\"\\\"" - S "%\\r" ) ^ 1 )
1361             +
1362             PercentInterpol
1363             +
1364             P "%"
1365             +
1366             EOL
1367         ) ^ 0
1368     )
1369     * Q ( P "\"\\\"" )
1370 )
1371 local LongString = SingleLongString + DoubleLongString

```

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`).

```

1372 local StringDoc =
1373     K ( 'String.Doc' , P "\"\\\"" )
1374     * ( K ( 'String.Doc' , ( 1 - P "\"\\\"" - P "\\r" ) ^ 0 ) * EOL
1375         * Tab ^ 0
1376     ) ^ 0
1377     * K ( 'String.Doc' , ( 1 - P "\"\\\"" - P "\\r" ) ^ 0 * P "\"\\\"" )

```

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

```

1378 local CommentMath =
1379     P "$" * K ( 'Comment.Math' , ( 1 - S "$\\r" ) ^ 1 ) * P "$"
1380
1381 local Comment =
1382     WithStyle ( 'Comment' ,
1383         Q ( P "#" )
1384         * ( CommentMath + Q ( ( 1 - S "$\\r" ) ^ 1 ) ) ^ 0 )
1385     * ( EOL + -1 )

```

The following LPEG `CommentLaTeX` is for what is called in that document the “LaTeX comments”. Since the elements that will be caught 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`).

```

1386 local CommentLaTeX =

```

```

1387 P(piton.comment_latex)
1388 * Lc "{\\PitonStyle{Comment.LaTeX}{\\ignorespaces}"
1389 * L ( ( 1 - P "\\r" ) ^ 0 )
1390 * Lc "}"
1391 * ( EOL + -1 )

```

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

```

1392 local expression =
1393   P { "E" ,
1394     E = ( P "'" * ( P "\\'" + 1 - S "'\\r" ) ^ 0 * P "'"
1395           + P "\"" * ( P "\\\"" + 1 - S "\"\\r" ) ^ 0 * P "\""
1396           + P "{" * V "F" * P "}"
1397           + P "(" * V "F" * P ")"
1398           + P "[" * V "F" * P "]"
1399           + ( 1 - S "{}() []\\r," ) ^ 0 ,
1400     F = ( P "{" * V "F" * P "}"
1401           + P "(" * V "F" * P ")"
1402           + P "[" * V "F" * P "]"
1403           + ( 1 - S "{}() []\\r\\'" ) ^ 0
1404   }

```

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**.

```

1405 local Param =
1406   SkipSpace * Identifier * SkipSpace
1407   * (
1408     K ( 'InitialValues' , P "=" * expression )
1409     + Q ( P ":" ) * SkipSpace * K ( 'Name.Type' , letter ^ 1 )
1410   ) ^ -1
1411 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**...

```

1412 local DefFunction =
1413   K ( 'Keyword' , P "def" )
1414   * Space
1415   * K ( 'Name.Function.Internal' , identifier )
1416   * SkipSpace
1417   * Q ( P "(" ) * Params * Q ( P ")" )
1418   * SkipSpace
1419   * ( 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.

```

1420 * K ( 'ParseAgain' , ( 1 - S "\\r" ) ^ 0 )
1421 * Q ( P ":" )

```

```

1422 * ( SkipSpace
1423     * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
1424     * Tab ^ 0
1425     * SkipSpace
1426     * StringDoc ^ 0 -- there may be additionnal docstrings
1427 ) ^ -1

```

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

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

**The dictionaries of Python** We have LPEG dealing with dictionaries of Python because, in typesettings of explicit Python dictionaries, one may prefer to have all the values formatted in black (in order to see more clearly the keys which are usually Python strings). That's why we have a `piton` style `Dict.Value`.

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

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

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

```

1428 local ItemDict =
1429     ShortString * SkipSpace * Q ( P ":" ) * K ( 'Dict.Value' , expression )
1430
1431 local ItemOfSet = SkipSpace * ( ItemDict + ShortString ) * SkipSpace
1432
1433 local Set =
1434     Q ( P "{" )
1435     * ItemOfSet * ( Q ( P "," ) * ItemOfSet ) ^ 0
1436     * Q ( P "}" )

```

## Miscellaneous

```
1437 local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL
```

**The main LPEG** First, the main loop :

```

1438 local MainPython =
1439     EOL
1440     + Space
1441     + Tab
1442     + Escape
1443     + CommentLaTeX
1444     + Beamer
1445     + LongString
1446     + Comment
1447     + ExceptionInConsole
1448     + Set
1449     + Delim
1450     + Operator
1451     + OperatorWord * ( Space + Punct + Delim + EOL + -1 )
1452     + ShortString
1453     + Punct
1454     + FromImport
1455     + RaiseException
1456     + DefFunction
1457     + DefClass
1458     + Keyword * ( Space + Punct + Delim + EOL + -1 )
1459     + Decorator
1460     + Builtin * ( Space + Punct + Delim + EOL + -1 )

```

```

1461     + Identifier
1462     + Number
1463     + Word

```

Ici, il ne faut pas mettre local !

```

1464 MainLoopPython =
1465   ( ( space^1 * -1 )
1466     + MainPython
1467   ) ^ 0

```

We recall that each line in the Python code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>27</sup>.

```

1468 local python = P ( true )
1469
1470 python =
1471   Ct (
1472     ( ( space - P "\r" ) ^0 * P "\r" ) ^ -1
1473     * BeamerBeginEnvironments
1474     * PromptHastyDetection
1475     * Lc '\\@@_begin_line:'
1476     * Prompt
1477     * SpaceIndentation ^ 0
1478     * MainLoopPython
1479     * -1
1480     * Lc '\\@@_end_line:'
1481   )
1482
1483 local languages = { }
1484 languages['python'] = python

```

### 6.3.3 The LPEG ocaml

```

1484 local Delim = Q ( P "[" + P "]" + S "[]" )
1485 local Punct = Q ( S ",:;! " )

```

The identifiers caught by `cap_identifier` begin with a cap. In OCaml, it's used for the constructors of types and for the modules.

```

1486 local cap_identifier = R "AZ" * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
1487 local Constructor = K ( 'Name.Constructor' , cap_identifier )
1488 local ModuleType = K ( 'Name.Type' , cap_identifier )

```

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

```

1489 local identifier =
1490   ( R "az" + P "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
1491 local Identifier = K ( 'Identifier' , identifier )

```

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

```

1492 local expression_for_fields =
1493   P { "E" ,
1494     E = ( P "{" * V "F" * P "}"
1495           + P "(" * V "F" * P ")"
1496           + P "[" * V "F" * P "]"
1497           + P "\"" * ( P "\\\"" + 1 - S "\"\r" ) ^0 * P "\""
1498           + P "'" * ( P "\\'" + 1 - S "'\r" ) ^0 * P "'"
1499           + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
1500     F = ( P "{" * V "F" * P "}"
1501           + P "(" * V "F" * P ")"
1502           + P "[" * V "F" * P "]"

```

---

<sup>27</sup>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:`

```

1503         + ( 1 - S "{ } ( [ ] \r \"'\" ) ) ^ 0
1504     }
1505 local OneFieldDefinition =
1506     ( K ( 'KeyWord' , P "mutable" ) * SkipSpace ) ^ -1
1507     * K ( 'Name.Field' , identifier ) * SkipSpace
1508     * Q ":" * SkipSpace
1509     * K ( 'Name.Type' , expression_for_fields )
1510     * SkipSpace
1511
1512 local OneField =
1513     K ( 'Name.Field' , identifier ) * SkipSpace
1514     * Q "=" * SkipSpace
1515     * ( C ( expression_for_fields ) / ( function (s) return LoopOCaml:match(s) end ) )
1516     * SkipSpace
1517
1518 local Record =
1519     Q "{" * SkipSpace
1520     *
1521     (
1522         OneFieldDefinition * ( Q ";" * SkipSpace * OneFieldDefinition ) ^ 0
1523         +
1524         OneField * ( Q ";" * SkipSpace * OneField ) ^ 0
1525     )
1526     *
1527     Q "}"

```

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.

```

1528 local DotNotation =
1529     (
1530         K ( 'Name.Module' , cap_identifier )
1531         * Q "."
1532         * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" )
1533
1534         +
1535         Identifier
1536         * Q "."
1537         * K ( 'Name.Field' , identifier )
1538     )
1539     * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0
1540 local Operator =
1541     K ( 'Operator' ,
1542         P "!=" + P "<>" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P "!="
1543         + P "||" + P "&&" + P "/" + P "/*" + P ";" + P "::" + P "->"
1544         + P "+." + P "-." + P ".*" + P "/"
1545         + S "-~+/*%=<>&@|"
1546     )
1547
1548 local OperatorWord =
1549     K ( 'Operator.Word' ,
1550         P "and" + P "asr" + P "land" + P "lor" + P "lsl" + P "lxor"
1551         + P "mod" + P "or" )
1552
1553 local Keyword =
1554     K ( 'Keyword' ,
1555         P "assert" + P "as" + P "begin" + P "class" + P "constraint" + P "done"
1556         + P "downto" + P "do" + P "else" + P "end" + P "exception" + P "external"
1557         + P "false" + P "for" + P "function" + P "functor" + P "fun" + P "if"
1558         + P "include" + P "inherit" + P "initializer" + P "in" + P "lazy" + P "let"
1559         + P "match" + P "method" + P "module" + P "mutable" + P "new" + P "object"
1560         + P "of" + P "open" + P "private" + P "raise" + P "rec" + P "sig"
1561         + P "struct" + P "then" + P "to" + P "true" + P "try" + P "type"
1562         + P "value" + P "val" + P "virtual" + P "when" + P "while" + P "with" )

```



```

1563   + K ( 'Keyword.Constant' , P "true" + P "false" )
1564
1565
1566 local Builtin =
1567   K ( 'Name.Builtin' , P "not" + P "incr" + P "decr" + P "fst" + P "snd" )

```

The following exceptions are exceptions in the standard library of OCaml (Stdlib).

```

1568 local Exception =
1569   K ( 'Exception' ,
1570     P "Division_by_zero" + P "End_of_File" + P "Failure"
1571   + P "Invalid_argument" + P "Match_failure" + P "Not_found"
1572   + P "Out_of_memory" + P "Stack_overflow" + P "Sys_blocked_io"
1573   + P "Sys_error" + P "Undefined_recursive_module" )

```

## The characters in OCaml

```

1574 local Char =
1575   K ( 'String.Short' , P "'" * ( ( 1 - P "'" ) ^ 0 + P "\\'" ) * P "'" )

```

## Beamer

```

1576 local balanced_braces =
1577   P { "E" ,
1578     E =
1579       (
1580         P "{" * V "E" * P "}"
1581       +
1582         P "\"" * ( 1 - S "\"" ) ^ 0 * P "\"" -- OCaml strings
1583       +
1584         ( 1 - S "{" )
1585       ) ^ 0
1586   }
1587
1588 if piton_beamer
1589 then
1590   Beamer =
1591     L ( P "\\pause" * ( P "[" * (1 - P "]") ^ 0 * P "]" ) ^ -1 )
1592   +
1593     ( P "\\uncover" * Lc ( '\\@@_beamer_command:n{uncover}' )
1594   + P "\\only" * Lc ( '\\@@_beamer_command:n{only}' )
1595   + P "\\alert" * Lc ( '\\@@_beamer_command:n{alert}' )
1596   + P "\\visible" * Lc ( '\\@@_beamer_command:n{visible}' )
1597   + P "\\invisible" * Lc ( '\\@@_beamer_command:n{invisible}' )
1598   + P "\\action" * Lc ( '\\@@_beamer_command:n{action}' )
1599   )
1600   *
1601   L ( ( P "<" * (1 - P ">") ^ 0 * P ">" ) ^ -1 * P "{" )
1602   * K ( 'ParseAgain.noCR' , balanced_braces )
1603   * L ( P "}" )
1604   +
1605   L (
1606     ( P "\\alt" )
1607     * P "<" * (1 - P ">") ^ 0 * P ">"
1608     * P "{"
1609   )
1610   * K ( 'ParseAgain.noCR' , balanced_braces )
1611   * L ( P "{" )
1612   * K ( 'ParseAgain.noCR' , balanced_braces )
1613   * L ( P "}" )
1614   +
1615   L (
1616     ( P "\\temporal" )
1617     * P "<" * (1 - P ">") ^ 0 * P ">"

```

```

1617         * P "{"
1618     )
1619     * K ( 'ParseAgain.noCR' , balanced_braces )
1620     * L ( P "}" )
1621     * K ( 'ParseAgain.noCR' , balanced_braces )
1622     * L ( P "}" )
1623     * K ( 'ParseAgain.noCR' , balanced_braces )
1624     * L ( P "}" )
1625     BeamerBeginEnvironments =
1626     ( space ^ 0 *
1627         L
1628         (
1629             P "\\begin{" * BeamerNamesEnvironments * "}"
1630             * ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1
1631         )
1632         * P "\r"
1633     ) ^ 0
1634     BeamerEndEnvironments =
1635     ( space ^ 0 *
1636         L ( P "\\end{" * BeamerNamesEnvironments * P "}" )
1637         * P "\r"
1638     ) ^ 0
1639 end

```

## EOL

```

1640 local EOL =
1641     P "\r"
1642     *
1643     (
1644         ( space^0 * -1 )
1645         +
1646         Ct (
1647             Cc "EOL"
1648             *
1649             Ct (
1650                 Lc "\\@@_end_line:"
1651                 * BeamerEndEnvironments
1652                 * BeamerBeginEnvironments
1653                 * PromptHastyDetection
1654                 * Lc "\\@@_newline: \\@@_begin_line:"
1655                 * Prompt
1656             )
1657         )
1658     )
1659     *
1660     SpaceIndentation ^ 0
1661 %
1662 % \paragraph{The strings}
1663 %
1664 % We need a pattern |string| without captures because it will be used within the
1665 % comments of OCaml.
1666 % \begin{macrocode}
1667 local ocaml_string =
1668     Q ( P "\"" )
1669     * (
1670         VisualSpace
1671         +
1672         Q ( ( 1 - S " \"\r" ) ^ 1 )
1673         +
1674         EOL
1675     ) ^ 0
1676     * Q ( P "\"" )

```

```
1677 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 programming, see the paragraphe *Lua's long strings* in [www.inf.puc-rio.br/~roberto/lpeg](http://www.inf.puc-rio.br/~roberto/lpeg).

```
1678 local ext = ( R "az" + P "_" ) ^ 0
1679 local open = "{" * Cg(ext, 'init') * "|"
1680 local close = "|" * C(ext) * "}"
1681 local closeeq =
1682   Cmt ( close * Cb('init'),
1683     function (s, i, a, b) return a==b end )
```

The LPEG QuotedStringBis will do the second analysis.

```
1684 local QuotedStringBis =
1685   WithStyle ( 'String.Long' ,
1686     (
1687       VisualSpace
1688       +
1689       Q ( ( 1 - S " \r" ) ^ 1 )
1690       +
1691       EOL
1692     ) ^ 0 )
1693
```

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.

```
1694 local QuotedString =
1695   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
1696   ( 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, therefore, the comments may 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).

```
1697 local Comment =
1698   WithStyle ( 'Comment' ,
1699     P {
1700       "A" ,
1701       A = Q "(" *
1702         * ( V "A"
1703           + Q ( ( 1 - P "(" - P ")" - S "\r$\\" ) ^ 1 ) -- $
1704           + ocaml_string
1705           + P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$" -- $
1706           + EOL
1707         ) ^ 0
1708       * Q ")"
1709     } )
```

## The DefFunction

```

1710 local balanced_parens =
1711   P { "E" ,
1712     E =
1713       (
1714         P "(" * V "E" * P ")"
1715         +
1716         ( 1 - S "(" )
1717       ) ^ 0
1718   }
1719 local Argument =
1720   K ( 'Identifier' , identifier )
1721   + Q "(" * SkipSpace
1722     * K ( 'Identifier' , identifier ) * SkipSpace
1723     * Q ":" * SkipSpace
1724     * K ( 'Name.Type' , balanced_parens ) * SkipSpace
1725     * Q ")"

```

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

```

1726 local DefFunction =
1727   K ( 'Keyword' , P "let open" )
1728   * Space
1729   * K ( 'Name.Module' , cap_identifier )
1730   +
1731   K ( 'Keyword' , P "let rec" + P "let" + P "and" )
1732   * Space
1733   * K ( 'Name.Function.Internal' , identifier )
1734   * Space
1735   * (
1736     Q "=" * SkipSpace * K ( 'Keyword' , P "function" )
1737     +
1738     Argument
1739     * ( SkipSpace * Argument ) ^ 0
1740     * (
1741       SkipSpace
1742       * Q ":"
1743       * K ( 'Name.Type' , ( 1 - P "=" ) ^ 0 )
1744     ) ^ -1
1745   )

```

**The DefModule** The following LPEG will be used in the definitions of modules but also in the definitions of *types* of modules.

```

1746 local DefModule =
1747   K ( 'Keyword' , P "module" ) * Space
1748   *
1749   (
1750     K ( 'Keyword' , P "type" ) * Space
1751     * K ( 'Name.Type' , cap_identifier )
1752   +
1753     K ( 'Name.Module' , cap_identifier ) * SkipSpace
1754     *
1755     (
1756       Q "(" * SkipSpace
1757       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
1758       * Q ":" * SkipSpace
1759       * K ( 'Name.Type' , cap_identifier ) * SkipSpace
1760       *
1761       (
1762         Q "," * SkipSpace
1763         * K ( 'Name.Module' , cap_identifier ) * SkipSpace
1764         * Q ":" * SkipSpace

```

```

1765         * K ( 'Name.Type' , cap_identifier ) * SkipSpace
1766     ) ^ 0
1767     * Q ")"
1768 ) ^ -1
1769 *
1770 (
1771     Q "=" * SkipSpace
1772     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
1773     * Q "("
1774     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
1775     *
1776     (
1777         Q ","
1778         *
1779         K ( 'Name.Module' , cap_identifier ) * SkipSpace
1780     ) ^ 0
1781     * Q ")"
1782 ) ^ -1
1783 )
1784 +
1785 K ( 'Keyword' , P "include" + P "open" )
1786 * Space * K ( 'Name.Module' , cap_identifier )

```

### The parameters of the types

```

1787 local TypeParameter = K ( 'TypeParameter' , P "'" * alpha * # ( 1 - P "'" ) )

```

### The main LPEG First, the main loop :

```

1788 MainOCaml =
1789     EOL
1790     + Space
1791     + Tab
1792     + Escape
1793     + Beamer
1794     + TypeParameter
1795     + String + QuotedString + Char
1796     + Comment
1797     + Delim
1798     + Operator
1799     + Punct
1800     + FromImport
1801     + Exception
1802     + DefFunction
1803     + DefModule
1804     + Record
1805     + Keyword * ( Space + Punct + Delim + EOL + -1 )
1806     + OperatorWord * ( Space + Punct + Delim + EOL + -1 )
1807     + Builtin * ( Space + Punct + Delim + EOL + -1 )
1808     + DotNotation
1809     + Constructor
1810     + Identifier
1811     + Number
1812     + Word
1813
1814 LoopOCaml = MainOCaml ^ 0
1815
1816 MainLoopOCaml =
1817     ( ( space^1 * -1 )
1818     + MainOCaml
1819     ) ^ 0

```

We recall that each line in the Python code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>28</sup>.

```

1820 local ocaml = P ( true )
1821
1822 ocaml =
1823   Ct (
1824     ( ( space - P "\r" ) ^0 * P "\r" ) ^ -1
1825     * BeamerBeginEnvironments
1826     * Lc ( '\\@@_begin_line:' )
1827     * SpaceIndentation ^ 0
1828     * MainLoopOCaml
1829     * -1
1830     * Lc ( '\\@@_end_line:' )
1831   )
1832 languages['ocaml'] = ocaml

```

### 6.3.4 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 python` which returns as capture a Lua table containing data to send to LaTeX.

```

1833 function piton.Parse(language,code)
1834   local t = languages[language] : match ( code )
1835   if t == nil
1836   then
1837     tex.sprint("\\PitonSyntaxError")
1838     return -- to exit in force the function
1839   end
1840   local left_stack = {}
1841   local right_stack = {}
1842   for _ , one_item in ipairs(t)
1843   do
1844     if one_item[1] == "EOL"
1845     then
1846       for _ , s in ipairs(right_stack)
1847       do tex.sprint(s)
1848       end
1849       for _ , s in ipairs(one_item[2])
1850       do tex.tprint(s)
1851       end
1852       for _ , s in ipairs(left_stack)
1853       do tex.sprint(s)
1854       end
1855     else

```

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 (`{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}`.

```

1856       if one_item[1] == "Open"
1857       then
1858         tex.sprint( one_item[2] )
1859         table.insert(left_stack,one_item[2])

```

---

<sup>28</sup>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:`

```

1860         table.insert(right_stack,one_item[3])
1861     else
1862         if one_item[1] == "Close"
1863         then
1864             tex.sprint( right_stack[#right_stack] )
1865             left_stack[#left_stack] = nil
1866             right_stack[#right_stack] = nil
1867         else
1868             tex.tprint(one_item)
1869         end
1870     end
1871 end
1872 end
1873 end

```

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

```

1874 function piton.ParseFile(language,name,first_line,last_line)
1875     local s = ''
1876     local i = 0
1877     for line in io.lines(name)
1878     do i = i + 1
1879         if i >= first_line
1880         then s = s .. '\r' .. line
1881         end
1882         if i >= last_line then break end
1883     end

```

We extract the BOM of utf-8, if present.

```

1884     if string.byte(s,1) == 13
1885     then if string.byte(s,2) == 239
1886         then if string.byte(s,3) == 187
1887             then if string.byte(s,4) == 191
1888                 then s = string.sub(s,5,-1)
1889             end
1890         end
1891     end
1892 end
1893 piton.Parse(language,s)
1894 end

```

### 6.3.5 Two variants of the function `Parse` with integrated preprocessors

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

```

1895 function piton.ParseBis(language,code)
1896     local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
1897     return piton.Parse(language,s)
1898 end

```

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 `\@@_piton:n` in the `piton` style of the syntactic element. In that case, you have to remove the potential `\@@_breakable_space:` that have been inserted when the key `break-lines` is in force.

```

1899 function piton.ParseTer(language,code)
1900     local s = ( Cs ( ( P '\\@@_breakable_space:' / ' ' + 1 ) ^ 0 ) )
1901             : match ( code )
1902     return piton.Parse(language,s)
1903 end

```

### 6.3.6 Preprocessors of the function Parse for gobble

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

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

```
1904 local function gobble(n,code)
1905     function concat(acc,new_value)
1906         return acc .. new_value
1907     end
1908     if n==0
1909     then return code
1910     else
1911         return Cf (
1912             Cc ( "" ) *
1913             ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
1914             * ( C ( P "\r" )
1915             * ( 1 - P "\r" ) ^ (-n)
1916             * C ( ( 1 - P "\r" ) ^ 0 )
1917             ) ^ 0 ,
1918             concat
1919         ) : match ( code )
1920     end
1921 end
```

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

```
1922 local function add(acc,new_value)
1923     return acc + new_value
1924 end
```

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

```
1925 local AutoGobbleLPEG =
1926     ( space ^ 0 * P "\r" ) ^ -1
1927     * Cf (
1928         (
```

We don't take into account the empty lines (with only spaces).

```
1929         ( P " " ) ^ 0 * P "\r"
1930         +
1931         Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1932         * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * P "\r"
1933     ) ^ 0
```

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

```
1934     *
1935     ( Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1936     * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,
1937     math.min
1938 )
```

The following LPEG is similar but works with the indentations.

```
1939 local TabsAutoGobbleLPEG =
1940     ( space ^ 0 * P "\r" ) ^ -1
1941     * Cf (
1942         (
1943             ( P "\t" ) ^ 0 * P "\r"
1944             +
1945             Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )
```



```

1946      * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * P "\r"
1947    ) ^ 0
1948    *
1949    ( Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )
1950    * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,
1951    math.min
1952  )

```

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

```

1953 local EnvGobbleLPEG =
1954   ( ( 1 - P "\r" ) ^ 0 * P "\r" ) ^ 0
1955   * Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add ) * -1

1956 function piton.GobbleParse(language,n,code)
1957   if n==1
1958     then n = AutoGobbleLPEG : match(code)
1959   else if n==2
1960     then n = EnvGobbleLPEG : match(code)
1961   else if n==3
1962     then n = TabsAutoGobbleLPEG : match(code)
1963   end
1964   end
1965   end
1966   piton.Parse(language,gobble(n,code))
1967 end

```

### 6.3.7 To count the number of lines

```

1968 function piton.CountLines(code)
1969   local count = 0
1970   for i in code : gmatch ( "\r" ) do count = count + 1 end
1971   tex.sprint(
1972     luatexbase.catcodetables.expl ,
1973     '\\int_set:Nn \\l_@@_nb_lines_int {' .. count .. '}' )
1974 end

1975 function piton.CountNonEmptyLines(code)
1976   local count = 0
1977   count =
1978   ( Cf ( Cc(0) *
1979     (
1980       ( P " " ) ^ 0 * P "\r"
1981       + ( 1 - P "\r" ) ^ 0 * P "\r" * Cc(1)
1982     ) ^ 0
1983     * ( 1 - P "\r" ) ^ 0 ,
1984     add
1985     ) * -1 ) : match (code)
1986   tex.sprint(
1987     luatexbase.catcodetables.expl ,
1988     '\\int_set:Nn \\l_@@_nb_non_empty_lines_int {' .. count .. '}' )
1989 end

1990 function piton.CountLinesFile(name)
1991   local count = 0
1992   for line in io.lines(name) do count = count + 1 end
1993   tex.sprint(
1994     luatexbase.catcodetables.expl ,

```

```

1995     '\int_set:Nn \l_@@_nb_lines_int {' .. count .. '}' )
1996 end

1997 function piton.CountNonEmptyLinesFile(name)
1998     local count = 0
1999     for line in io.lines(name)
2000     do if not ( ( ( P " " ) ^ 0 * -1 ) : match ( line ) )
2001         then count = count + 1
2002     end
2003     end
2004     tex.sprint(
2005         luatexbase.catcodetables.expl ,
2006         '\int_set:Nn \l_@@_nb_non_empty_lines_int {' .. count .. '}' )
2007 end
2008 \end{luacode*}

```

## 7 History

The successive versions of the file `piton.sty` provided by TeXLive are available on the SVN server of TeXLive:

<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:

<https://github.com/fpantigny/piton>

### 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 keys `break-lines-in-piton` and `break-lines-in-Piton`.

New key `show-spaces-in-string` and modification of the key `show-spaces`.

When the class beamer is used, the environements `{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

<b>1</b>	<b>Presentation</b>	<b>1</b>
<b>2</b>	<b>Use of the package</b>	<b>2</b>
2.1	Loading the package . . . . .	2
2.2	The tools provided to the user . . . . .	2
2.3	The syntax of the command <code>\piton</code> . . . . .	2
<b>3</b>	<b>Customization</b>	<b>3</b>
3.1	The command <code>\PitonOptions</code> . . . . .	3
3.2	The styles . . . . .	5
3.3	Creation of new environments . . . . .	6

<b>4</b>	<b>Advanced features</b>	<b>7</b>
4.1	Highlighting some identifiers	7
4.2	Mechanisms to escape to LaTeX	8
4.2.1	The “LaTeX comments”	8
4.2.2	The key “math-comments”	9
4.2.3	The mechanism “escape-inside”	9
4.3	Behaviour in the class Beamer	10
4.3.1	{Piton} et \PitonInputFile are “overlay-aware”	11
4.3.2	Commands of Beamer allowed in {Piton} and \PitonInputFile	11
4.3.3	Environments of Beamer allowed in {Piton} and \PitonInputFile	12
4.4	Page breaks and line breaks	12
4.4.1	Page breaks	12
4.4.2	Line breaks	13
4.5	Footnotes in the environments of piton	14
4.6	Tabulations	14
<b>5</b>	<b>Examples</b>	<b>14</b>
5.1	Line numbering	14
5.2	Formatting of the LaTeX comments	15
5.3	Notes in the listings	16
5.4	An example of tuning of the styles	17
5.5	Use with pyluatex	17
<b>6</b>	<b>Implementation</b>	<b>20</b>
6.1	Introduction	20
6.2	The L3 part of the implementation	21
6.2.1	Declaration of the package	21
6.2.2	Parameters and technical definitions	23
6.2.3	Treatment of a line of code	26
6.2.4	PitonOptions	29
6.2.5	The numbers of the lines	31
6.2.6	The command to write on the aux file	31
6.2.7	The main commands and environments for the final user	31
6.2.8	The styles	37
6.2.9	The initial styles	39
6.2.10	Highlighting some identifiers	40
6.2.11	Security	41
6.2.12	The error messages of the package	41
6.3	The Lua part of the implementation	42
6.3.1	Special functions dealing with LPEG	43
6.3.2	The LPEG python	45
6.3.3	The LPEG ocaml	55
6.3.4	The function Parse	62
6.3.5	Two variants of the function Parse with integrated preprocessors	63
6.3.6	Preprocessors of the function Parse for gobble	64
6.3.7	To count the number of lines	65
<b>7</b>	<b>History</b>	<b>66</b>