

The package `piton`*

F. Pantigny
fpantigny@wanadoo.fr

April 4, 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 LPEG¹ 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 1.5 of `piton`, at the date of 2023/04/04.

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

²This 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 commands³ are fully expanded and not executed,
so it's possible to use `\\` to insert a backslash.

³That concerns the commands beginning with a backslash but also the active characters.

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

Examples :

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

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

- **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 :

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

3 Customization

3.1 The command `\PitonOptions`

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

- 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.
- **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.

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

⁵We remind that a LaTeX environment is, in particular, a TeX group.

- 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 13.

- The key `background-color` sets the background color of the environments `{Piton}` and the listings produced by `\PitonInputFile` (that background has a width of `\linewidth`).

New 1.4 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*).
- 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.⁶

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`⁷ 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)

    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}
```

⁶The 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`.

⁷cf. 4.4.2 p. 12

```

1 from math import pi
2
3 def arctan(x,n=10):
4     """Compute the mathematical value of arctan(x)
5
6     n is the number of terms in the sum
7     """
8     if x < 0:
9         return -arctan(-x) # recursive call
10    elif x > 1:
11        return pi/2 - arctan(1/x)
12        (we have used that  $\arctan(x) + \arctan(1/x) = \frac{\pi}{2}$  for  $x > 0$ )
13    else
14        s = 0
15        for k in range(n):
16            s += (-1)**k/(2*k+1)*x**(2*k+1)
17    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.⁸

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`

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`.⁹

New 1.4 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*.

⁸We remind that a LaTeX environment is, in particular, a TeX group.

⁹See: <https://pygments.org/styles/>. Remark that, by default, Pygments provides for its style `manni` a colored background whose color is the HTML color `#F0F3F3`. It’s possible to have the same color in `{Pion}` with the instruction `\PitonOptions{background-color = [HTML]{F0F3F3}}`.

3.3 Creation of new environments

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

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

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

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

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

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

```
\NewPitonEnvironment{Python}{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

New 1.4 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}`.

```

\PytonOptions
{
  identifiers =
  {
    names = { l1 , l2 } ,
    style = \color{red}
  }
}

\begin{Pyton}
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{Pyton}

```

```

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 `pyton`.

```

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

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

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

For example, with the following loading:

```
\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. 14

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.¹⁰

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`.

¹⁰That 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 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`).

```
\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`.¹¹

When the package `piton` is used within the class `beamer`¹², the behaviour of `piton` is slightly modified, as described now.

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`¹³ ;
- 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 strings¹⁴ of Python are not considered.

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"""
```

¹¹Remind 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).

¹²The 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}`

¹³One 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

¹⁴The 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.

```

\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).

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.¹⁵

4.4.2 Line breaks

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

- With the key `break-lines-in-piton`, the line breaks are allowed in the command `\piton{...}` (but not in the command `\piton|...|`, that is to say the command `\piton` in verbatim mode).
- With the key `break-lines-in-Piton`, the line breaks are allowed in the environment `{Piton}` (hence the capital letter P in the name) and in the listings produced by `\PitonInputFile`.
- The key `break-lines` is a conjunction of the two previous keys.

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

- With the key `indent-broken-lines`, the indentation of a broken line is respected at carriage return.
- The key `end-of-broken-line` corresponds to the symbol placed at the end of a broken line. The initial value is: `\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 in a standard LaTeX `{minipage}` of width 12 cm with the following tuning:

```
\PitonOptions{break-lines,indent-broken-lines,background-color=gray!15}
```

¹⁵With the key `splittable`, the environments `{Piton}` are breakable, even within a (breakable) environment of `tcolorbox`. Remind that an environment of `tcolorbox` included in another environment of `tcolorbox` is *not* breakable, even when both environments use the key `breakable` of `tcolorbox`.

```

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

```

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. 15.

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).

```

\PytonOptions{background-color=gray!10, left-margin = auto, line-numbers}
\begin{Pyton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)          #> (appel récursif)
    elif x > 1:
        return pi/2 - arctan(1/x) #> (autre appel récursif)
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Pyton}

```

```

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

```

5.2 Formatting of the LaTeX comments

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

```

\PytonOptions{background-color=gray!10}
\SetPytonStyle{Comment.LaTeX = \hfill \normalfont\color{gray}}
\begin{Pyton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)          #> appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x) #> autre appel récursif
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Pyton}

```

```

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

```

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

```

\PytonOptions{background-color=gray!10}
\NewDocumentCommand{\MyLaTeXCommand}{m}{\hfill \normalfont\itshape\rlap{\quad #1}}
\SetPytonStyle{Comment.LaTeX = \MyLaTeXCommand}
\begin{minipage}{12cm}
\begin{Pyton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)          #> appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x) #> autre appel récursif
    else:
        s = 0
        for k in range(n):

```

```

        s += (-1)**k/(2*k+1)*x**(2*k+1)
    return s
\end{Piton}
\end{minipage}

```

```

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

```

appel récursif

autre appel récursif

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. 13. 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)16
    elif x > 1:
        return pi/2 - arctan(1/x)17
    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:

```

¹⁶First recursive call.

¹⁷Second recursive call.

```

        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) )

```

^aFirst recursive call.

^bSecond recursive call.

If we embed an environment `{Piton}` in an environment `{minipage}` (typically in order to limit the width of a colored background), it's necessary to embed the whole environment `{minipage}` in an environment `{savenotes}` (of `footnote` or `footnotehyper`) in order to have the footnotes composed at the bottom of the page.

```

\PitonOptions{background-color=gray!10}
\begin{savenotes}
\begin{minipage}{13cm}
\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}
\end{savenotes}

```

```

def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)18
    elif x > 1:
        return pi/2 - arctan(1/x)19
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )

```

5.4 An example of tuning of the styles

The graphical styles have been presented in the section 3.2, p. 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*²⁰ 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}

```

¹⁸First recursive call.

¹⁹Second recursive call.

²⁰See: <https://dejavu-fonts.github.io>

```

\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) # appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x)
        (we have used that arctan(x) + arctan(1/x) =  $\pi/2$  for  $x > 0$ )
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s

```

5.5 Use with pyluatex

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

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

```

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

```

```

        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>!= == << >> - ~ + / * % = < > & . @</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>#></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

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.²¹

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:" }
```

^aEach 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`).

^bThe 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`.

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

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`)

²¹Recall 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.


```

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

```
120 \dim_new:N \g_@@_width_dim
```

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

```
121 \dim_new:N \l_@@_width_on_aux_dim
```

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

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

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

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

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

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

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

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

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

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

```
128 % The following token list corresponds to the key
```

```
129 % |continuation-symbol-on-indentation|. The name has been shorten to |csoi|.
```

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

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

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

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

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

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

```
134 \bool_new:N \l_@@_break_lines_in_piton_bool
```

The following boolean corresponds to the key `slim` of `\PitonOptions`.

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

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

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

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

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

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

```
138 \dim_new:N \l_@@_numbers_sep_dim
```

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

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

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

```
141 \cs_new_protected:Npn \@@_set_tab_tl:n #1
```

```
142 {
```

```
143   \tl_clear:N \l_@@_tab_tl
```

```
144   \prg_replicate:nn { #1 }
```

```
145     { \tl_put_right:Nn \l_@@_tab_tl { ~ } }
```

```
146 }
```

```
147 \@@_set_tab_tl:n { 4 }
```

The following integer corresponds to the key `gobble`.

```

148 \int_new:N \l_@@_gobble_int

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

```

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

```

151 \int_new:N \g_@@_indentation_int

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

180 \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.

```

181 \cs_new_protected:Npn \@@_prompt:
182 {
183   \tl_gset:Nn \g_@@_begin_line_hook_tl
184   { \clist_set:NV \l_@@_bg_color_clist \l_@@_prompt_bg_color_tl }
185 }

```

You will keep track of the current style for the treatment of EOL (for the multi-line syntactic elements).

```

186 \clist_new:N \g_@@_current_style_clist
187 \clist_set:Nn \g_@@_current_style_clist { __end }

```

The element `__end` is an arbitrary syntactic marker.

```

188 \cs_new_protected:Npn \@@_close_current_styles:
189 {
190   \int_set:Nn \l_tmpa_int { \clist_count:N \g_@@_current_style_clist - 1 }
191   \exp_args:NV \@@_close_n_styles:n \l_tmpa_int
192 }
193 \cs_new_protected:Npn \@@_close_n_styles:n #1
194 {
195   \int_compare:nNnT { #1 } > 0
196   {
197     \@@_close_brace:
198     \@@_close_brace:
199     \@@_close_n_styles:n { #1 - 1 }
200   }
201 }
202 \cs_new_protected:Npn \@@_open_current_styles:
203 { \exp_last_unbraced:NV \@@_open_styles:w \g_@@_current_style_clist , }
204 \cs_new_protected:Npn \@@_open_styles:w #1 ,
205 {
206   \tl_if_eq:nnF { #1 } { __end }
207   { \@@_open_brace: #1 \@@_open_brace: \@@_open_styles:w }
208 }
209 \cs_new_protected:Npn \@@_pop_style:
210 {
211   \clist_greverse:N \g_@@_current_style_clist
212   \clist_gpop:NN \g_@@_current_style_clist \l_tmpa_tl
213   \clist_gpop:NN \g_@@_current_style_clist \l_tmpa_tl
214   \clist_gpush:Nn \g_@@_current_style_clist { __end }
215   \clist_greverse:N \g_@@_current_style_clist
216 }
217 \cs_new_protected:Npn \@@_push_style:n #1
218 {
219   \clist_greverse:N \g_@@_current_style_clist
220   \clist_gpop:NN \g_@@_current_style_clist \l_tmpa_tl
221   \clist_gpush:Nn \g_@@_current_style_clist { #1 }
222   \clist_gpush:Nn \g_@@_current_style_clist { __end }
223   \clist_greverse:N \g_@@_current_style_clist
224 }
225 \cs_new_protected:Npn \@@_push_and_exec:n #1
226 {
227   \@@_push_style:n { #1 }
228   \@@_open_brace: #1 \@@_open_brace:
229 }

```

6.2.3 Treatment of a line of code

```

230 \cs_new_protected:Npn \@@_replace_spaces:n #1
231 {
232   \tl_set:Nn \l_tmpa_tl { #1 }
233   \bool_if:NTF \l_@@_show_spaces_bool
234     { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
235     {

```

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.

```

236     \bool_if:NT \l_@@_break_lines_in_Piton_bool
237     {
238       \regex_replace_all:nnN
239         { \x20 }
240         { \c { @@_breakable_space: } }
241       \l_tmpa_tl
242     }
243   }
244   \l_tmpa_tl
245 }
246 \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.

```

247 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
248 {
249   \group_begin:
250   \g_@@_begin_line_hook_tl
251   \int_gzero:N \g_@@_indentation_int

```

Be careful: there is curryfication in the following lines.

```

252   \bool_if:NTF \l_@@_slim_bool
253     { \hcoffin_set:Nn \l_tmpa_coffin }
254     {
255       \clist_if_empty:NTF \l_@@_bg_color_clist
256         {
257           \vcoffin_set:Nnn \l_tmpa_coffin
258             { \dim_eval:n { \linewidth - \l_@@_left_margin_dim } }
259         }
260         {
261           \vcoffin_set:Nnn \l_tmpa_coffin
262             { \dim_eval:n { \linewidth - \l_@@_left_margin_dim - 0.5 em } }
263         }
264       }
265     {
266       \language = -1
267       \raggedright
268       \strut
269       \@@_replace_spaces:n { #1 }
270       \strut \hfil
271     }
272   \hbox_set:Nn \l_tmpa_box
273   {
274     \skip_horizontal:N \l_@@_left_margin_dim
275     \bool_if:NT \l_@@_line_numbers_bool
276     {
277       \bool_if:NF \l_@@_all_line_numbers_bool
278       { \tl_if_empty:nF { #1 } }
279       \@@_print_number:
280     }

```

```

281 \clist_if_empty:NF \l_@@_bg_color_clist
282 {
283   \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
284   {
285     \bool_if:NF \l_@@_left_margin_auto_bool
286     { \skip_horizontal:n { 0.5 em } }
287   }
288 }
289 \coffin_typeset:Nnnnn \l_tmpa_coffin T l \c_zero_dim \c_zero_dim
290 }

```

We compute in `\g_@@_width_dim` the maximal width of the lines of the environment.

```

291 \dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_width_dim
292 { \dim_gset:Nn \g_@@_width_dim { \box_wd:N \l_tmpa_box } }
293 \box_set_dp:Nn \l_tmpa_box { \box_dp:N \l_tmpa_box + 1.25 pt }
294 \box_set_ht:Nn \l_tmpa_box { \box_ht:N \l_tmpa_box + 1.25 pt }
295 \clist_if_empty:NTF \l_@@_bg_color_clist
296 { \box_use_drop:N \l_tmpa_box }
297 {
298   \vbox_top:n
299   {
300     \hbox:n
301     {
302       \@@_color:N \l_@@_bg_color_clist
303       \vrule height \box_ht:N \l_tmpa_box
304         depth \box_dp:N \l_tmpa_box
305         width \l_@@_width_on_aux_dim
306     }
307     \skip_vertical:n { - \box_ht_plus_dp:N \l_tmpa_box }
308     \box_set_wd:Nn \l_tmpa_box \l_@@_width_on_aux_dim
309     \box_use_drop:N \l_tmpa_box
310   }
311 }
312 \vspace { - 2.5 pt }
313 \group_end:
314 \tl_gclear:N \g_@@_begin_line_hook_tl
315 }

```

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).

```

316 \cs_set_protected:Npn \@@_color:N #1
317 {
318   \int_set:Nn \l_tmpa_int { \clist_count:N #1 }
319   \int_set:Nn \l_tmpb_int { \int_mod:nn \g_@@_line_int \l_tmpa_int + 1 }
320   \tl_set:Nx \l_tmpa_tl { \clist_item:Nn #1 \l_tmpb_int }
321   \tl_if_eq:NnTF \l_tmpa_tl { none }

```

By setting `\l_@@_width_on_aux_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).

```

322   { \dim_zero:N \l_@@_width_on_aux_dim }
323   { \exp_args:NV \@@_color_i:n \l_tmpa_tl }
324 }

```

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} }`.

```

325 \cs_set_protected:Npn \@@_color_i:n #1
326 {
327   \tl_if_head_eq_meaning:nNTF { #1 } [
328     {
329       \tl_set:Nn \l_tmpa_tl { #1 }
330       \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
331       \exp_last_unbraced:NV \color \l_tmpa_tl
332     }
333     { \color { #1 } }

```

```

334 }
335 \cs_generate_variant:Nn \@@_color:n { V }

336 \cs_new_protected:Npn \@@_newline:
337 {
338   \int_gincr:N \g_@@_line_int
339   \int_compare:nNnT \g_@@_line_int > { \l_@@_splittable_int - 1 }
340   {
341     \int_compare:nNnT
342       { \l_@@_nb_lines_int - \g_@@_line_int } > \l_@@_splittable_int
343       {
344         \egroup
345         \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
346         \newline
347         \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
348         \vtop \bgroup
349       }
350   }
351 }

352 \cs_set_protected:Npn \@@_breakable_space:
353 {
354   \discretionary
355     { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
356     {
357       \hbox_overlap_left:n
358         {
359           {
360             \normalfont \footnotesize \color { gray }
361             \l_@@_continuation_symbol_tl
362           }
363           \skip_horizontal:n { 0.3 em }
364           \clist_if_empty:NF \l_@@_bg_color_clist
365             { \skip_horizontal:n { 0.5 em } }
366         }
367       \bool_if:NT \l_@@_indent_broken_lines_bool
368       {
369         \hbox:n
370           {
371             \prg_replicate:nn { \g_@@_indentation_int } { ~ }
372             { \color { gray } \l_@@_csoi_tl }
373           }
374       }
375     }
376   { \hbox { ~ } }
377 }

```

6.2.4 PitonOptions

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

```

378 \bool_new:N \l_@@_line_numbers_bool
379 \bool_new:N \l_@@_all_line_numbers_bool

```

The following flag corresponds to the key `resume`.

```

380 \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.

```

381 \keys_define:nn { PitonOptions }
382 {
383   language          .str_set:N          = \l_@@_language_str ,

```

```

384 language .value_required:n = true ,
385 gobble .int_set:N = \l_@@_gobble_int ,
386 gobble .value_required:n = true ,
387 auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
388 auto-gobble .value_forbidden:n = true ,
389 env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
390 env-gobble .value_forbidden:n = true ,
391 tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
392 tabs-auto-gobble .value_forbidden:n = true ,
393 line-numbers .bool_set:N = \l_@@_line_numbers_bool ,
394 line-numbers .default:n = true ,
395 all-line-numbers .code:n =
396   \bool_set_true:N \l_@@_line_numbers_bool
397   \bool_set_true:N \l_@@_all_line_numbers_bool ,
398 all-line-numbers .value_forbidden:n = true ,
399 resume .bool_set:N = \l_@@_resume_bool ,
400 resume .value_forbidden:n = true ,
401 splittable .int_set:N = \l_@@_splittable_int ,
402 splittable .default:n = 1 ,
403 background-color .clist_set:N = \l_@@_bg_color_clist ,
404 background-color .value_required:n = true ,
405 prompt-background-color .tl_set:N = \l_@@_prompt_bg_color_tl ,
406 prompt-background-color .value_required:n = true ,
407 slim .bool_set:N = \l_@@_slim_bool ,
408 slim .default:n = true ,
409 left-margin .code:n =
410   \str_if_eq:nnTF { #1 } { auto }
411   {
412     \dim_zero:N \l_@@_left_margin_dim
413     \bool_set_true:N \l_@@_left_margin_auto_bool
414   }
415   { \dim_set:Nn \l_@@_left_margin_dim { #1 } } ,
416 left-margin .value_required:n = true ,
417 numbers-sep .dim_set:N = \l_@@_numbers_sep_dim ,
418 numbers-sep .value_required:n = true ,
419 tab-size .code:n = \@@_set_tab_tl:n { #1 } ,
420 tab-size .value_required:n = true ,
421 show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
422 show-spaces .default:n = true ,
423 show-spaces-in-strings .code:n = \tl_set:Nn \l_@@_space_tl { \_ } , % U+2423
424 show-spaces-in-strings .value_forbidden:n = true ,
425 break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
426 break-lines-in-Piton .default:n = true ,
427 break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
428 break-lines-in-piton .default:n = true ,
429 break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
430 break-lines .value_forbidden:n = true ,
431 indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,
432 indent-broken-lines .default:n = true ,
433 end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
434 end-of-broken-line .value_required:n = true ,
435 continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
436 continuation-symbol .value_required:n = true ,
437 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
438 continuation-symbol-on-indentation .value_required:n = true ,
439 unknown .code:n =
440   \msg_error:nn { piton } { Unknown-key-for-PitonOptions }
441 }

```

The argument of `\PitonOptions` is provided by curryfication.

```

442 \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`).

```
443 \int_new:N \g_@@_visual_line_int
444 \cs_new_protected:Npn \@@_print_number:
445 {
446   \int_gincr:N \g_@@_visual_line_int
447   \hbox_overlap_left:n
448   {
449     { \color { gray } \footnotesize \int_to_arabic:n \g_@@_visual_line_int }
450     \skip_horizontal:N \l_@@_numbers_sep_dim
451   }
452 }
```

6.2.6 The command to write on the aux file

```
453 \cs_new_protected:Npn \@@_write_aux:
454 {
455   \tl_if_empty:NF \g_@@_aux_tl
456   {
457     \iow_now:Nn \@mainaux { \ExplSyntaxOn }
458     \iow_now:Nx \@mainaux
459     {
460       \tl_gset:cn { c_@@_ \int_use:N \g_@@_env_int _ tl }
461       { \exp_not:V \g_@@_aux_tl }
462     }
463     \iow_now:Nn \@mainaux { \ExplSyntaxOff }
464   }
465   \tl_gclear:N \g_@@_aux_tl
466 }
467 \cs_new_protected:Npn \@@_width_to_aux:
468 {
469   \bool_if:NT \l_@@_slim_bool
470   {
471     \clist_if_empty:NF \l_@@_bg_color_clist
472     {
473       \tl_gput_right:Nx \g_@@_aux_tl
474       {
475         \dim_set:Nn \l_@@_width_on_aux_dim
476         { \dim_eval:n { \g_@@_width_dim + 0.5 em } }
477       }
478     }
479   }
480 }
```

6.2.7 The main commands and environments for the final user

```
481 \NewDocumentCommand { \piton } { }
482 { \peek_meaning:NTF \bgroup \@@_piton_standard \@@_piton_verbatim }
483 \NewDocumentCommand { \@@_piton_standard } { m }
484 {
485   \group_begin:
486   \ttfamily
```

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

```
487 \automatichyphenmode = 1
488 \cs_set_eq:NN \ \c_backslash_str
489 \cs_set_eq:NN \% \c_percent_str
```

```

490 \cs_set_eq:NN \{ \c_left_brace_str
491 \cs_set_eq:NN \} \c_right_brace_str
492 \cs_set_eq:NN \$ \c_dollar_str
493 \cs_set_eq:cN { ~ } \space
494 \cs_set_protected:Npn \@@_begin_line: { }
495 \cs_set_protected:Npn \@@_end_line: { }
496 \tl_set:Nx \l_tmpa_tl
497 {
498   \lua_now:e
499   { piton.ParseBis('\l_@@_language_str',token.scan_string()) }
500   { #1 }
501 }
502 \bool_if:NTF \l_@@_show_spaces_bool
503 { \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.

```

504 {
505   \bool_if:NT \l_@@_break_lines_in_piton_bool
506   { \regex_replace_all:nnN { \x20 } { \x20 } \l_tmpa_tl }
507 }
508 \l_tmpa_tl
509 \group_end:
510 }
511 \NewDocumentCommand { \@@_piton_verbatim } { v }
512 {
513   \group_begin:
514   \ttfamily
515   \automaticshyphenmode = 1
516   \cs_set_protected:Npn \@@_begin_line: { }
517   \cs_set_protected:Npn \@@_end_line: { }
518   \tl_set:Nx \l_tmpa_tl
519   {
520     \lua_now:e
521     { piton.Parse('\l_@@_language_str',token.scan_string()) }
522     { #1 }
523   }
524   \bool_if:NT \l_@@_show_spaces_bool
525   { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
526   \l_tmpa_tl
527   \group_end:
528 }

```

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).

```

529 \cs_new_protected:Npn \@@_piton:n #1
530 {
531   \group_begin:
532   \cs_set_protected:Npn \@@_begin_line: { }
533   \cs_set_protected:Npn \@@_end_line: { }
534   \bool_lazy_or:nnTF
535   \l_@@_break_lines_in_piton_bool
536   \l_@@_break_lines_in_Piton_bool
537   {
538     \tl_set:Nx \l_tmpa_tl
539     {
540       \lua_now:e
541       { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
542       { #1 }
543     }
544   }

```

```

545     {
546         \tl_set:Nx \l_tmpa_tl
547         {
548             \lua_now:e
549             { piton.Parse('\l_@@_language_str',token.scan_string()) }
550             { #1 }
551         }
552     }
553     \bool_if:NT \l_@@_show_spaces_bool
554     { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
555     \l_tmpa_tl
556     \group_end:
557 }

```

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

```

558 \cs_new_protected:Npn \@@_piton_no_cr:n #1
559 {
560     \group_begin:
561     \cs_set_protected:Npn \@@_begin_line: { }
562     \cs_set_protected:Npn \@@_end_line: { }
563     \cs_set_protected:Npn \@@_newline:
564     { \msg_fatal:nn { piton } { cr~not~allowed } }
565     \bool_lazy_or:nnTF
566     \l_@@_break_lines_in_piton_bool
567     \l_@@_break_lines_in_Piton_bool
568     {
569         \tl_set:Nx \l_tmpa_tl
570         {
571             \lua_now:e
572             { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
573             { #1 }
574         }
575     }
576     {
577         \tl_set:Nx \l_tmpa_tl
578         {
579             \lua_now:e
580             { piton.Parse('\l_@@_language_str',token.scan_string()) }
581             { #1 }
582         }
583     }
584     \bool_if:NT \l_@@_show_spaces_bool
585     { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
586     \l_tmpa_tl
587     \group_end:
588 }

```

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

```

589 \cs_new:Npn \@@_pre_env:
590 {
591     \automatichyphenmode = 1
592     \int_gincr:N \g_@@_env_int
593     \tl_gclear:N \g_@@_aux_tl
594     \cs_if_exist_use:c { c_@@ _ \int_use:N \g_@@_env_int _ tl }
595     \dim_compare:nNnT \l_@@_width_on_aux_dim = \c_zero_dim
596     { \dim_set_eq:NN \l_@@_width_on_aux_dim \linewidth }
597     \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
598     \dim_gzero:N \g_@@_width_dim
599     \int_gzero:N \g_@@_line_int
600     \dim_zero:N \parindent
601     \dim_zero:N \lineskip

```

```

602 \dim_zero:N \parindent
603 \cs_set_eq:NN \label \@@_label:n
604 }

605 \keys_define:nn { PitonInputFile }
606 {
607   first-line .int_set:N = \l_@@_first_line_int ,
608   first-line .value_required:n = true ,
609   last-line .int_set:N = \l_@@_last_line_int ,
610   last-line .value_required:n = true ,
611 }

612 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
613 {
614   \tl_if_novalue:nF { #1 }
615   {
616     \bool_if:NTF \c_@@_beamer_bool
617     { \begin { uncoverenv } < #1 > }
618     { \msg_error:nn { piton } { overlay~without~beamer } }
619   }
620   \group_begin:
621     \int_zero_new:N \l_@@_first_line_int
622     \int_zero_new:N \l_@@_last_line_int
623     \int_set_eq:NN \l_@@_last_line_int \c_max_int
624     \keys_set:nn { PitonInputFile } { #2 }
625     \@@_pre_env:
626     \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.

```

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

```

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

```

628 \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
629 {
630   \hbox_set:Nn \l_tmpa_box
631   {
632     \footnotesize
633     \bool_if:NTF \l_@@_all_line_numbers_bool
634     {
635       \int_to_arabic:n
636       { \g_@@_visual_line_int + \l_@@_nb_lines_int }
637     }
638     {
639       \lua_now:n
640       { piton.CountNonEmptyLinesFile(token.scan_argument()) }
641       { #3 }
642       \int_to_arabic:n
643       { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
644     }
645   }
646   \dim_set:Nn \l_@@_left_margin_dim
647   { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
648 }

```

Now, the main job.

```

649 \ttfamily
650 \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
651 \vtop \bgroup
652 \lua_now:e
653 {
654   piton.ParseFile('\l_@@_language_str',token.scan_argument() ,

```

```

655         \int_use:N \l_@@_first_line_int ,
656         \int_use:N \l_@@_last_line_int )
657     }
658     { #3 }
659     \egroup
660     \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
661     \@@_width_to_aux:
662     \group_end:
663     \tl_if_novalue:nF { #1 }
664     { \bool_if:NT \c_@@_beamer_bool { \end { uncoverenv } } }
665     \@@_write_aux:
666 }

667 \NewDocumentCommand { \NewPitonEnvironment } { m m m m }
668 {

```

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.

```

669     \use:x
670     {
671         \cs_set_protected:Npn
672         \use:c { _@@_collect_ #1 :w }
673         #####1
674         \c_backslash_str end \c_left_brace_str #1 \c_right_brace_str
675     }
676     {
677         \group_end:
678         \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.

```

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

```

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

```

680         \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
681         {
682             \bool_if:NTF \l_@@_all_line_numbers_bool
683             {
684                 \hbox_set:Nn \l_tmpa_box
685                 {
686                     \footnotesize
687                     \int_to_arabic:n
688                     { \g_@@_visual_line_int + \l_@@_nb_lines_int }
689                 }
690             }
691             {
692                 \lua_now:n
693                 { piton.CountNonEmptyLines(token.scan_argument()) }
694                 { ##1 }
695                 \hbox_set:Nn \l_tmpa_box
696                 {
697                     \footnotesize
698                     \int_to_arabic:n
699                     { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
700                 }
701             }
702             \dim_set:Nn \l_@@_left_margin_dim
703             { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
704         }

```

Now, the main job.

```

705         \ttfamily
706         \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }

```

```

707     \vtop \bgroup
708     \lua_now:e
709     {
710         piton.GobbleParse
711         (
712             '\l_@@_language_str' ,
713             \int_use:N \l_@@_gobble_int ,
714             token.scan_argument()
715         )
716     }
717     { ##1 }
718     \vspace { 2.5 pt }
719     \egroup
720     \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
721     \@@_width_to_aux:

```

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

```

722     \end { #1 }
723     \@@_write_aux:
724 }

```

We can now define the new environment.

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

```

725     \NewDocumentEnvironment { #1 } { #2 }
726     {
727         #3
728         \@@_pre_env:
729         \group_begin:
730         \tl_map_function:nN
731         { \ \ \ \ { \ } \ $ \ & \ # \ ^ \ _ \ % \ ~ \ ^ \ I }
732         \char_set_catcode_other:N
733         \use:c { _@@_collect_ #1 :w }
734     }
735     { #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).

```

736     \AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^M }
737 }

```

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`.

```

738 \bool_if:NTF \c_@@_beamer_bool
739 {
740     \NewPitonEnvironment { Piton } { d < > }
741     {
742         \IfValueTF { #1 }
743         { \begin { uncoverenv } < #1 > }
744         { \begin { uncoverenv } }
745     }
746     { \end { uncoverenv } }
747 }
748 { \NewPitonEnvironment { Piton } { } { } { } { } }

```

6.2.8 The styles

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

```

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

```

The following command takes in its argument by curryfication.

```

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

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

753 \keys_define:nn { piton / Styles }
754   {
755     String.Interpol .tl_set:c = pitonStyle String.Interpol ,
756     String.Interpol .value_required:n = true ,
757     FormattingType .tl_set:c = pitonStyle FormattingType ,
758     FormattingType .value_required:n = true ,
759     Dict.Value .tl_set:c = pitonStyle Dict.Value ,
760     Dict.Value .value_required:n = true ,
761     Name.Decorator .tl_set:c = pitonStyle Name.Decorator ,
762     Name.Decorator .value_required:n = true ,
763     Name.Function .tl_set:c = pitonStyle Name.Function ,
764     Name.Function .value_required:n = true ,
765     Name.UserFunction .tl_set:c = pitonStyle Name.UserFunction ,
766     Name.UserFunction .value_required:n = true ,
767     Keyword .tl_set:c = pitonStyle Keyword ,
768     Keyword .value_required:n = true ,
769     Keyword.Constant .tl_set:c = pitonStyle Keyword.Constant ,
770     Keyword.constant .value_required:n = true ,
771     String.Doc .tl_set:c = pitonStyle String.Doc ,
772     String.Doc .value_required:n = true ,
773     Interpol.Inside .tl_set:c = pitonStyle Interpol.Inside ,
774     Interpol.Inside .value_required:n = true ,
775     String.Long .tl_set:c = pitonStyle String.Long ,
776     String.Long .value_required:n = true ,
777     String.Short .tl_set:c = pitonStyle String.Short ,
778     String.Short .value_required:n = true ,
779     String .meta:n = { String.Long = #1 , String.Short = #1 } ,
780     Comment.Math .tl_set:c = pitonStyle Comment.Math ,
781     Comment.Math .default:n = \@@_math_scantokens:n ,
782     Comment.Math .initial:n = ,
783     Comment .tl_set:c = pitonStyle Comment ,
784     Comment .value_required:n = true ,
785     InitialValues .tl_set:c = pitonStyle InitialValues ,
786     InitialValues .value_required:n = true ,
787     Number .tl_set:c = pitonStyle Number ,
788     Number .value_required:n = true ,
789     Name.Namespace .tl_set:c = pitonStyle Name.Namespace ,
790     Name.Namespace .value_required:n = true ,
791     Name.Class .tl_set:c = pitonStyle Name.Class ,
792     Name.Class .value_required:n = true ,
793     Name.Builtin .tl_set:c = pitonStyle Name.Builtin ,
794     Name.Builtin .value_required:n = true ,
795     TypeParameter .tl_set:c = pitonStyle TypeParameter ,
796     TypeParameter .value_required:n = true ,
797     Name.Type .tl_set:c = pitonStyle Name.Type ,
798     Name.Type .value_required:n = true ,
799     Operator .tl_set:c = pitonStyle Operator ,
800     Operator .value_required:n = true ,
801     Operator.Word .tl_set:c = pitonStyle Operator.Word ,
802     Operator.Word .value_required:n = true ,
803     Exception .tl_set:c = pitonStyle Exception ,
804     Exception .value_required:n = true ,
805     Comment.LaTeX .tl_set:c = pitonStyle Comment.LaTeX ,
806     Comment.LaTeX .value_required:n = true ,
807     Identifier .tl_set:c = pitonStyle Identifier ,
808     Comment.LaTeX .value_required:n = true ,
809     ParseAgain.noCR .tl_set:c = pitonStyle ParseAgain.noCR ,

```

```

810 ParseAgain.noCR .value_required:n = true ,
811 ParseAgain .tl_set:c = pitonStyle ParseAgain ,
812 ParseAgain .value_required:n = true ,
813 Prompt .tl_set:c = pitonStyle Prompt ,
814 Prompt .value_required:n = true ,
815 unknown .code:n =
816 \msg_error:nn { piton } { Unknown-key-for-SetPitonStyle }
817 }

818 \msg_new:nnn { piton } { Unknown-key-for-SetPitonStyle }
819 {
820 The~style~'\l_keys_key_str'~is~unknown.\\
821 This~key~will~be~ignored.\\
822 The~available~styles~are~(in~alphabetic~order):~
823 Comment,~
824 Comment.LaTeX,~
825 Dict.Value,~
826 Exception,~
827 Identifier,~
828 InitialValues,~
829 Keyword,~
830 Keyword.Constant,~
831 Name.Builtin,~
832 Name.Class,~
833 Name.Decorator,~
834 Name.Function,~
835 Name.Namespace,~
836 Number,~
837 Operator,~
838 Operator.Word,~
839 Prompt,~
840 String,~
841 String.Doc,~
842 String.Long,~
843 String.Short,~and~
844 String.Interpol.
845 }

```

6.2.9 The initial style

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

```

846 \SetPitonStyle
847 {
848 Comment = \color[HTML]{0099FF} \itshape ,
849 Exception = \color[HTML]{CC0000} ,
850 Keyword = \color[HTML]{006699} \bfseries ,
851 Keyword.Constant = \color[HTML]{006699} \bfseries ,
852 Name.Builtin = \color[HTML]{336666} ,
853 Name.Decorator = \color[HTML]{9999FF},
854 Name.Class = \color[HTML]{00AA88} \bfseries ,
855 Name.Function = \color[HTML]{CC00FF} ,
856 Name.Namespace = \color[HTML]{00CCFF} ,
857 Number = \color[HTML]{FF6600} ,
858 Operator = \color[HTML]{555555} ,
859 Operator.Word = \bfseries ,
860 String = \color[HTML]{CC3300} ,
861 String.Doc = \color[HTML]{CC3300} \itshape ,
862 String.Interpol = \color[HTML]{AA0000} ,
863 Comment.LaTeX = \normalfont \color[rgb]{.468,.532,.6} ,
864 Name.Type = \color[HTML]{336666} ,

```

```

865 InitialValues      = \@@_piton:n ,
866 Dict.Value         = \@@_piton:n ,
867 Interpol.Inside    = \color{black}\@@_piton:n ,
868 TypeParameter      = \color[HTML]{008800} \itshape ,
869 Identifier          = \@@_identifier:n ,
870 Name.UserFunction   = ,
871 Prompt             = ,
872 ParseAgain.noCR     = \@@_piton_no_cr:n ,
873 ParseAgain          = \@@_piton:n ,
874 }

```

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)].

```

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

```

6.2.10 Highlighting some identifiers

```

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

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

880 \keys_define:nn { Piton / identifiers }
881 {
882   names .clist_set:N = \l_@@_identifiers_names_tl ,
883   style .tl_set:N    = \l_@@_style_tl ,
884 }

885 \cs_new_protected:Npn \@@_set_identifiers:n #1
886 {
887   \clist_clear_new:N \l_@@_identifiers_names_tl
888   \tl_clear_new:N \l_@@_style_tl
889   \keys_set:nn { Piton / identifiers } { #1 }
890   \clist_map_inline:Nn \l_@@_identifiers_names_tl
891   {
892     \tl_set_eq:cN
893     { PitonIdentifier _ \l_@@_language_str _ ##1 }
894     \l_@@_style_tl
895   }
896 }

```

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`).

```

897 \cs_new_protected:cpn { pitonStyle Name.Function.Internal } #1
898 {

```

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

```

899   { \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 `Name.UserFunction`. Of course, here the affectation is global because we have to exit many groups and even the environments `{Piton}`).

```

900 \cs_gset_protected:cpn { PitonIdentifier _ \l_@@_language_str _ #1 }
901 { \PitonStyle{ Name.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.

```

902 \seq_if_exist:cF { g_@@_functions _ \l_@@_language_str _ seq }
903 { \seq_new:c { g_@@_functions _ \l_@@_language_str _ seq } }
904 \seq_gput_right:cn { g_@@_functions _ \l_@@_language_str _ seq } { #1 }
905 }

```

```

906 \NewDocumentCommand \PitonClearUserFunctions { ! 0 { \l_@@_language_str } }
907 {
908   \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
909   {
910     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
911     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1} }
912     \seq_gclear:c { g_@@_functions _ #1 _ seq }
913   }
914 }

```

6.2.11 Security

```

915 \AddToHook { env / piton / begin }
916 { \msg_fatal:nn { piton } { No-environment~piton } }
917
918 \msg_new:nnn { piton } { No-environment~piton }
919 {
920   There-is~no-environment~piton!\\
921   There-is~an-environment~{Piton}~and~a~command~
922   \token_to_str:N \piton\ but~there-is~no-environment~
923   {piton}.~This~error-is~fatal.
924 }

```

6.2.12 The error messages of the package

```

925 \msg_new:nnnn { piton } { Unknown-key~for~PitonOptions }
926 {
927   Unknown~key. \\
928   The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
929   It~will~be~ignored.\\
930   For~a~list~of~the~available~keys,~type~H~<return>.
931 }
932 {
933   The~available~keys~are~(in~alphabetic~order):~
934   all-line-numbers,~
935   auto-gobble,~
936   background-color,~
937   break-lines,~
938   break-lines-in-piton,~
939   break-lines-in-Piton,~
940   continuation-symbol,~
941   continuation-symbol-on-indentation,~
942   end-of-broken-line,~
943   env-gobble,~
944   gobble,~
945   identifiers,~
946   indent-broken-lines,~
947   language,~
948   left-margin,~
949   line-numbers,~
950   prompt-background-color,~
951   resume,~
952   show-spaces,~
953   show-spaces-in-strings,~

```

```

954     slim,~
955     splittable,~
956     tabs-auto-gobble~
957     and~tab-size.
958 }

959 \msg_new:nnn { piton } { label~with~lines~numbers }
960 {
961     You~can't~use~the~command~\token_to_str:N \label\
962     because~the~key~'line-numbers'~(or~'all-line-numbers')~
963     is~not~active.\\
964     If~you~go~on,~that~command~will~ignored.
965 }

966 \msg_new:nnn { piton } { cr-not~allowed }
967 {
968     You~can't~put~any~carriage~return~in~the~argument~
969     of~a~command~\c_backslash_str
970     \l_@@_beamer_command_str\ within~an~
971     environment~of~'piton'.~You~should~consider~using~the~
972     corresponding~environment.\\
973     That~error~is~fatal.
974 }

975 \msg_new:nnn { piton } { overlay~without~beamer }
976 {
977     You~can't~use~an~argument~<...>~for~your~command~
978     \token_to_str:N \PitonInputFile\ because~you~are~not~
979     in~Beamer.\\
980     If~you~go~on,~that~argument~will~be~ignored.
981 }

982 \msg_new:nnn { Piton } { Python~error }
983 { A~Python~error~has~been~detected. }

```

6.3 The Lua part of the implementation

```

984 \ExplSyntaxOff
985 \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`.

```

986 \begin{luacode*}
987 piton = piton or {}

988 if piton.comment_latex == nil then piton.comment_latex = ">" end
989 piton.comment_latex = "#" .. piton.comment_latex

```

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

```

990 function piton.open_brace ()
991     tex.sprint("{")
992 end
993 function piton.close_brace ()
994     tex.sprint("}")
995 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.

```
996 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
997 local Cf, Cs, Cg, Cmt, Cb = lpeg.Cf, lpeg.Cs, lpeg.Cg, lpeg.Cmt, lpeg.Cb
998 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”).

```
999 local function Q(pattern)
1000     return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
1001 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.

```
1002 local function L(pattern)
1003     return Ct ( C ( pattern ) )
1004 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.

```
1005 local function Lc(string)
1006     return Cc ( { luatexbase.catcodetables.expl, string } )
1007 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)

```
1008 local function K(style, pattern)
1009     return
1010         Lc ( "{\\PitonStyle{" .. style .. "}}{" )
1011         * Q ( pattern )
1012         * Lc ( "}" )
1013 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}}`.

```
1014 local function WithStyle(style, pattern)
1015     return
1016         Ct ( Cc "Open" * Cc ( "{\\PitonStyle{" .. style .. "}}{" ) * Cc "}" )
1017         * pattern
1018         * Ct ( Cc "Close" )
1019 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`²². 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.

```
1020 local Escape =
1021   P(piton_begin_escape)
1022   * L ( ( 1 - P(piton_end_escape) ) ^ 1 )
1023   * P(piton_end_escape)
```

The following line is mandatory.

```
1024 lpeg.locale(lpeg)
```

The basic syntactic LPEG

```
1025 local alpha, digit = lpeg.alpha, lpeg.digit
1026 local space = P " "
```

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

```
1027 local letter = alpha + P "_"
1028   + P "â" + P "à" + P "ç" + P "é" + P "è" + P "ê" + P "ë" + P "ï" + P "î"
1029   + P "ô" + P "û" + P "ü" + P "Â" + P "Ã" + P "Ç" + P "É" + P "Ê" + P "Ë"
1030   + P "Ï" + P "Î" + P "Ï" + P "Ô" + P "Õ" + P "Ü"
1031
1032 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).

```
1033 local identifier = letter * alphanum ^ 0
```

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

```
1034 local Identifier = K ( 'Identifier' , identifier)
```

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

Here is the first use of our function `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).

```
1035 local Number =
1036   K ( 'Number' ,
1037     ( digit^1 * P "." * digit^0 + digit^0 * P "." * digit^1 + digit^1 )
1038     * ( S "eE" * S "+-" ^ -1 * digit^1 ) ^ -1
1039     + digit^1
1040   )
```

We recall that `piton.begin_espace` and `piton_end_escape` are Lua strings corresponding to the key `escape-inside`²³. Of course, if the final user has not used the key `escape-inside`, these strings are empty.

```
1041 local Word
```

²²The `piton` key `escape-inside` is available at load-time only.

²³The `piton` key `escape-inside` is available at load-time only.

```

1042 if piton_begin_escape ~= ''
1043 then Word = Q ( ( ( 1 - space - P(piton_begin_escape) - P(piton_end_escape) )
1044               - S "'\"\\r[()]" - digit ) ^ 1 )
1045 else Word = Q ( ( ( 1 - space ) - S "'\"\\r[()]" - digit ) ^ 1 )
1046 end

1047 local Space = ( Q " " ) ^ 1
1048
1049 local SkipSpace = ( Q " " ) ^ 0
1050
1051 local Punct = Q ( S ".,:;!" )
1052
1053 local Tab = P "\t" * Lc ( '\\l_@@_tab_t1' )

1054 local SpaceIndentation = Lc ( '\\l_@@_an_indentation_space:' ) * ( Q " " )

1055 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.

```

1056 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.

```

1057 local Operator =
1058   K ( 'Operator' ,
1059       P "!=" + P "<>" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P "!="
1060       + P "/" + P "*" + S "-~/*%=<>&.@|"
1061   )
1062
1063 local OperatorWord =
1064   K ( 'Operator.Word' , P "in" + P "is" + P "and" + P "or" + P "not" )
1065
1066 local Keyword =
1067   K ( 'Keyword' ,
1068       P "as" + P "assert" + P "break" + P "case" + P "class" + P "continue"
1069       + P "def" + P "del" + P "elif" + P "else" + P "except" + P "exec"
1070       + P "finally" + P "for" + P "from" + P "global" + P "if" + P "import"
1071       + P "lambda" + P "non local" + P "pass" + P "return" + P "try"
1072       + P "while" + P "with" + P "yield" + P "yield from" )
1073   + K ( 'Keyword.Constant' , P "True" + P "False" + P "None" )
1074
1075 local Builtin =
1076   K ( 'Name.Builtin' ,
1077       P "__import__" + P "abs" + P "all" + P "any" + P "bin" + P "bool"
1078       + P "bytearray" + P "bytes" + P "chr" + P "classmethod" + P "compile"
1079       + P "complex" + P "delattr" + P "dict" + P "dir" + P "divmod"
1080       + P "enumerate" + P "eval" + P "filter" + P "float" + P "format"
1081       + P "frozenset" + P "getattr" + P "globals" + P "hasattr" + P "hash"
1082       + P "hex" + P "id" + P "input" + P "int" + P "isinstance" + P "issubclass"
1083       + P "iter" + P "len" + P "list" + P "locals" + P "map" + P "max"
1084       + P "memoryview" + P "min" + P "next" + P "object" + P "oct" + P "open"
1085       + P "ord" + P "pow" + P "print" + P "property" + P "range" + P "repr"

```

```

1086 + P "reversed" + P "round" + P "set" + P "setattr" + P "slice" + P "sorted"
1087 + P "staticmethod" + P "str" + P "sum" + P "super" + P "tuple" + P "type"
1088 + P "vars" + P "zip" )
1089
1090
1091 local Exception =
1092   K ( 'Exception' ,
1093     P "ArithmeticError" + P "AssertionError" + P "AttributeError"
1094   + P "BaseException" + P "BufferError" + P "BytesWarning" + P "DeprecationWarning"
1095   + P "EOFError" + P "EnvironmentError" + P "Exception" + P "FloatingPointError"
1096   + P "FutureWarning" + P "GeneratorExit" + P "IOError" + P "ImportError"
1097   + P "ImportWarning" + P "IndentationError" + P "IndexError" + P "KeyError"
1098   + P "KeyboardInterrupt" + P "LookupError" + P "MemoryError" + P "NameError"
1099   + P "NotImplementedError" + P "OSError" + P "OverflowError"
1100   + P "PendingDeprecationWarning" + P "ReferenceError" + P "ResourceWarning"
1101   + P "RuntimeError" + P "RuntimeWarning" + P "StopIteration"
1102   + P "SyntaxError" + P "SyntaxWarning" + P "SystemError" + P "SystemExit"
1103   + P "TabError" + P "TypeError" + P "UnboundLocalError" + P "UnicodeDecodeError"
1104   + P "UnicodeEncodeError" + P "UnicodeError" + P "UnicodeTranslateError"
1105   + P "UnicodeWarning" + P "UserWarning" + P "ValueError" + P "VMSError"
1106   + P "Warning" + P "WindowsError" + P "ZeroDivisionError"
1107   + P "BlockingIOError" + P "ChildProcessError" + P "ConnectionError"
1108   + P "BrokenPipeError" + P "ConnectionAbortedError" + P "ConnectionRefusedError"
1109   + P "ConnectionResetError" + P "FileExistsError" + P "FileNotFoundError"
1110   + P "InterruptedError" + P "IsADirectoryError" + P "NotADirectoryError"
1111   + P "PermissionError" + P "ProcessLookupError" + P "TimeoutError"
1112   + P "StopAsyncIteration" + P "ModuleNotFoundError" + P "RecursionError" )
1113
1114
1115 local RaiseException = K ( 'Keyword' , P "raise" ) * SkipSpace * Exception * Q ( P "(" )
1116

```

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

```

1117 local Decorator = K ( 'Name.Decorator' , P "@" * letter1 )

```

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:`

```

1118 local DefClass =
1119   K ( 'Keyword' , P "class" ) * Space * K ( 'Name.Class' , identifier )

```

If the word `class` is not followed by a 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 piton style Name.Namespace.

Example: `import numpy as np`

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

Example: `import math, numpy`

```

1120 local ImportAs =
1121   K ( 'Keyword' , P "import" )
1122   * Space
1123   * K ( 'Name.Namespace' ,
1124     identifier * ( P "." * identifier ) 0 )
1125   * (
1126     ( Space * K ( 'Keyword' , P "as" ) * Space
1127     * K ( 'Name.Namespace' , identifier ) )
1128   +

```

```

1129         ( SkipSpace * Q ( P "," ) * SkipSpace
1130         * K ( 'Name.Namespace' , identifier ) ) ^ 0
1131     )

```

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

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

Example: `from math import pi`

```

1132 local FromImport =
1133     K ( 'Keyword' , P "from" )
1134     * Space * K ( 'Name.Namespace' , identifier )
1135     * 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'''	"""text"""

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 instruction²⁴ in that interpolation:

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

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

```

1136 local PercentInterpol =
1137     K ( 'String.Interpol' ,
1138         P "%"
1139         * ( P "(" * alphanum ^ 1 * P ")" ) ^ -1
1140         * ( S "-#0 +" ) ^ 0
1141         * ( digit ^ 1 + P "*" ) ^ -1
1142         * ( P "." * ( digit ^ 1 + P "*" ) ) ^ -1
1143         * ( S "HLL" ) ^ -1
1144         * S "sdFeExXorgiGauc%"
1145     )

```

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.²⁵

```

1146 local SingleShortString =
1147     WithStyle ( 'String.Short' ,

```

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

```

1148     Q ( P "f" + P "F" )
1149     * (
1150         K ( 'String.Interpol' , P "{" )
1151         * K ( 'Interpol.Outside' , ( 1 - S "}'" ) ^ 0 )
1152         * Q ( P ":" * ( 1 - S "}'" ) ^ 0 ) ^ -1

```

²⁴There is no special `piton` style for the formatting instruction (after the colon): the style which will be applied will be the style of the encompassing string, that is to say `String.Short` or `String.Long`.

²⁵The interpolations are formatted with the `piton` style `Interpol.Outside`. The initial value of that style is `@@@_piton:n` which means that the interpolations are parsed once again by `piton`.

```

1153         * K ( 'String.Interpol' , P "}" )
1154     +
1155     VisualSpace
1156     +
1157     Q ( ( P "\\'" + P "{" + P "}" ) + 1 - S " {'" ) ^ 1 )
1158 ) ^ 0
1159 * Q ( P "'" )
1160 +

```

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

```

1161     Q ( P "'" + P "r'" + P "R'" )
1162     * ( Q ( ( P "\\'" + 1 - S " '\r%" ) ^ 1 )
1163         + VisualSpace
1164         + PercentInterpol
1165         + Q ( P "%" )
1166     ) ^ 0
1167     * Q ( P "'" ) )
1168
1169
1170 local DoubleShortString =
1171     WithStyle ( 'String.Short' ,
1172         Q ( P "f\"" + P "F\"" )
1173         * (
1174             K ( 'String.Interpol' , P "{" )
1175             * Q ( ( 1 - S "}\"") ^ 0 , 'Interpol.Inside' )
1176             * ( K ( 'String.Interpol' , P ":" ) * Q ( ( 1 - S "}:\\"" ) ^ 0 ) ) ^ -1
1177             * K ( 'String.Interpol' , P "}" )
1178         +
1179         VisualSpace
1180         +
1181         Q ( ( P "\\\"" + P "{" + P "}" ) + 1 - S " {\\"" ) ^ 1 )
1182     ) ^ 0
1183     * Q ( P "\" )
1184 +
1185     Q ( P "\" + P "r\"" + P "R\"" )
1186     * ( Q ( ( P "\\\"" + 1 - S " \"\r%" ) ^ 1 )
1187         + VisualSpace
1188         + PercentInterpol
1189         + Q ( P "%" )
1190     ) ^ 0
1191     * Q ( P "\" ) )
1192
1193 local ShortString = SingleShortString + DoubleShortString

```

Beamer The following LPEG `BalancedBraces` 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).

```

1194 local BalancedBraces =
1195     P { "E" ,
1196         E =
1197         (
1198             P "{" * V "E" * P "}"
1199             +
1200             ShortString
1201             +
1202             ( 1 - S "{" )
1203         ) ^ 0
1204     }

```

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

```

1205 local Beamer = P ( false )
1206 local BeamerBeginEnvironments = P ( true )
1207 local BeamerEndEnvironments = P ( true )
1208 local BeamerNamesEnvironments =
1209   P "uncoverenv" + P "onlyenv" + P "visibleenv" + P "invisibleenv"
1210   + P "alertenv" + P "actionenv"
1211
1212 UserCommands =
1213   Ct ( Cc "Open" * C ( "\\emph{" ) * Cc "}" )
1214   * ( C ( BalancedBraces ) / (function (s) return MainLoopPython:match(s) end ) )
1215   * P "}" * Ct ( Cc "Close" )
1216
1217 function OneBeamerEnvironment(name)
1218   return
1219   Ct ( Cc "Open"
1220     * C (
1221       P ( "\\begin{" .. name .. "}" )
1222       * ( P "<" * (1 - P ">") ^ 0 * P ">" ) ^ -1
1223     )
1224     * Cc ( "\\end{" .. name .. "}" )
1225   )
1226   * (
1227     C ( (1 - P ( "\\end{" .. name .. "}" ) ) ^ 0 )
1228     / (function (s) return MainLoopPython:match(s) end )
1229   )
1230   * P ( "\\end{" .. name .. "}" ) * Ct ( Cc "Close" )
1231 end
1232
1233 if piton_beamer
1234 then
1235   Beamer =
1236   L ( P "\\pause" * ( P "[" * (1 - P "]") ^ 0 * P "]" ) ^ -1 )
1237   +
1238   Ct ( Cc "Open"
1239     * C (
1240       (
1241         P "\\uncover" + P "\\only" + P "\\alert" + P "\\visible"
1242         + P "\\invisible" + P "\\action"
1243       )
1244       * ( P "<" * (1 - P ">") ^ 0 * P ">" ) ^ -1
1245       * P "{"
1246     )
1247     * Cc "}"
1248   )
1249   * ( C ( BalancedBraces ) / (function (s) return MainLoopPython:match(s) end ) )
1250   * P "}" * Ct ( Cc "Close" )
1251   +
1252   OneBeamerEnvironment "uncoverenv"
1253   + OneBeamerEnvironment "onlyenv"
1254   + OneBeamerEnvironment "visibleenv"
1255   + OneBeamerEnvironment "invisibleenv"
1256   + OneBeamerEnvironment "alertenv"
1257   + OneBeamerEnvironment "actionenv"
1258   +
1259   L (
1260     ( P "\\alt" )
1261     * P "<" * (1 - P ">") ^ 0 * P ">"
1262     * P "{"
1263   )
1264   * K ( 'ParseAgain.noCR' , BalancedBraces )
1265   * L ( P "}" )

```

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

```

1264     * K ( 'ParseAgain.noCR' , BalancedBraces )
1265     * L ( P "}" )
1266   +
1267     L (

```

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

```

1268       ( P "\\temporal" )
1269       * P "<" * ( 1 - P ">" ) ^ 0 * P ">"
1270       * P "{"
1271     )
1272     * K ( 'ParseAgain.noCR' , BalancedBraces )
1273     * L ( P "}" )
1274     * K ( 'ParseAgain.noCR' , BalancedBraces )
1275     * L ( P "}" )
1276     * K ( 'ParseAgain.noCR' , BalancedBraces )
1277     * L ( P "}" )

```

Now for the environments.

```

1278   BeamerBeginEnvironments =
1279     ( space ^ 0 *
1280       L
1281         (
1282           P "\\begin{" * BeamerNamesEnvironments * "}"
1283           * ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1
1284         )
1285       * P "\r"
1286     ) ^ 0
1287   BeamerEndEnvironments =
1288     ( space ^ 0 *
1289       L ( P "\\end{" * BeamerNamesEnvironments * P "}" )
1290       * P "\r"
1291     ) ^ 0
1292 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 the background).

```

1293 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).

```

1294 local Prompt = K ( 'Prompt' , ( ( P ">>>" + P "..." ) * P " " ^ -1 ) ^ -1 )

```

The following LPEG EOL is for the end of lines.

```

1295 local EOL =
1296   P "\r"
1297   *
1298   (
1299     ( space^0 * -1 )
1300   +

```

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:`²⁶.

²⁶Remember 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:`

```

1301 Ct (
1302     Cc "EOL"
1303     *
1304     Ct (
1305         Lc "\\@@_end_line:"
1306         * BeamerEndEnvironments
1307         * BeamerBeginEnvironments
1308         * PromptHastyDetection
1309         * Lc "\\@@_newline: \\@@_begin_line:"
1310         * Prompt
1311     )
1312 )
1313 )
1314 *
1315 SpaceIndentation ^ 0

```

The long strings

```

1316 local SingleLongString =
1317   WithStyle ( 'String.Long' ,
1318     ( Q ( S "fF" * P "'''" )
1319       * (
1320         K ( 'String.Interpol' , P "{" )
1321         * K ( 'Interpol.Inside' , ( 1 - S "};\r" - P "'''" ) ^ 0 )
1322         * Q ( P ":" * ( 1 - S "};\r" - P "'''" ) ^ 0 ) ^ -1
1323         * K ( 'String.Interpol' , P "}" )
1324         +
1325         Q ( ( 1 - P "'''" - S "{'}\r" ) ^ 1 )
1326         +
1327         EOL
1328       ) ^ 0
1329     +
1330     Q ( ( S "rR" ) ^ -1 * P "'''" )
1331     * (
1332       Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
1333       +
1334       PercentInterpol
1335       +
1336       P "%"
1337       +
1338       EOL
1339     ) ^ 0
1340   )
1341   * Q ( P "'''" ) )
1342
1343
1344 local DoubleLongString =
1345   WithStyle ( 'String.Long' ,
1346     (
1347       Q ( S "fF" * P "\"\"\"" )
1348       * (
1349         K ( 'String.Interpol', P "{" )
1350         * K ( 'Interpol.Inside' , ( 1 - S "};\r" - P "\"\"\"" ) ^ 0 )
1351         * Q ( P ":" * ( 1 - S "};\r" - P "\"\"\"" ) ^ 0 ) ^ -1
1352         * K ( 'String.Interpol' , P "}" )
1353         +
1354         Q ( ( 1 - P "\"\"\"" - S "{'}\r" ) ^ 1 )
1355         +
1356         EOL
1357       ) ^ 0
1358     +
1359     Q ( ( S "rR" ) ^ -1 * P "\"\"\"" )
1360     * (

```

```

1361         Q ( ( 1 - P "\"\"\"" - S "%\r" ) ^ 1 )
1362         +
1363         PercentInterpol
1364         +
1365         P "%"
1366         +
1367         EOL
1368     ) ^ 0
1369 )
1370 * Q ( P "\"\"\"" )
1371 )
1372 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`).

```

1373 local StringDoc =
1374     K ( 'String.Doc' , P "\"\"\"" )
1375     * ( K ( 'String.Doc' , ( 1 - P "\"\"\"" - P "\r" ) ^ 0 ) * EOL
1376     * Tab ^ 0
1377     ) ^ 0
1378     * 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.

```

1379 local CommentMath =
1380     P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$"
1381
1382 local Comment =
1383     WithStyle ( 'Comment' ,
1384         Q ( P "#" )
1385         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 )
1386     * ( 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`).

```

1387 local CommentLaTeX =
1388     P(piton.comment_latex)
1389     * Lc "{\\PitonStyle{Comment.LaTeX}}{\\ignorespaces}"
1390     * L ( ( 1 - P "\r" ) ^ 0 )
1391     * Lc "}"
1392     * ( EOL + -1 ) -- you could put EOL instead of EOL

```

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).

```

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

```

```

1403         P "'" * (P "\\'" + 1 - S"'\\r" )^0 * P "'"
1404     + P "\"" * (P "\\\"" + 1 - S"\"\\r" )^0 * P "\""
1405     + P "{" * V "F" * P "}"
1406     + P "(" * V "F" * P ")"
1407     + P "[" * V "F" * P "]"
1408     ) * ( 1 - S "{}() []\\r\\'" ) ^ 0 ) ^ 0 ,
1409 }

```

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.

```

1410 local Param =
1411     SkipSpace * Identifier * SkipSpace
1412     * (
1413         K ( 'InitialValues' , P "=" * Expression )
1414         + Q ( P ":" ) * SkipSpace * K ( 'Name.Type' , letter^1 )
1415     ) ^ -1
1416 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`...

```

1417 local DefFunction =
1418     K ( 'Keyword' , P "def" )
1419     * Space
1420     * K ( 'Name.Function.Internal' , identifier )
1421     * SkipSpace
1422     * Q ( P "(" ) * Params * Q ( P ")" )
1423     * SkipSpace
1424     * ( 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.

```

1425 * K ( 'ParseAgain' , ( 1 - S ":\r" )^0 )
1426 * Q ( P ":" )
1427 * ( SkipSpace
1428     * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
1429     * Tab ^ 0
1430     * SkipSpace
1431     * StringDoc ^ 0 -- there may be additionnal docstrings
1432 ) ^ -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!

```
1433 local ItemDict =
1434   ShortString * SkipSpace * Q ( P ":" ) * K ( 'Dict.Value' , Expression )
1435
1436 local ItemOfSet = SkipSpace * ( ItemDict + ShortString ) * SkipSpace
1437
1438 local Set =
1439   Q ( P "{" )
1440   * ItemOfSet * ( Q ( P "," ) * ItemOfSet ) ^ 0
1441   * Q ( P "}" )
```

Miscellaneous

```
1442 local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL
```

The main LPEG First, the main loop :

```
1443 MainLoopPython =
1444   ( ( space^1 * -1 )
1445     + EOL
1446     + Space
1447     + Tab
1448     + Escape
1449     + CommentLaTeX
1450     + Beamer
1451     + UserCommands
1452     + LongString
1453     + Comment
1454     + ExceptionInConsole
1455     + Set
1456     + Delim
```

Operator must be before Punct.

```
1457     + Operator
1458     + ShortString
1459     + Punct
1460     + FromImport
1461     + RaiseException
1462     + DefFunction
1463     + DefClass
1464     + Keyword * ( Space + Punct + Delim + EOL+ -1 )
1465     + Decorator
1466     + OperatorWord * ( Space + Punct + Delim + EOL+ -1 )
1467     + Builtin * ( Space + Punct + Delim + EOL+ -1 )
1468     + Identifier
1469     + Number
1470     + Word
1471   ) ^ 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:`²⁷.

```

1472 local python = P ( true )
1473
1474 python =
1475   Ct (
1476     ( ( space - P "\r" ) ^0 * P "\r" ) ^ -1
1477     * BeamerBeginEnvironments
1478     * PromptHastyDetection
1479     * Lc '\\@@_begin_line:'
1480     * Prompt
1481     * SpaceIndentation ^ 0
1482     * MainLoopPython
1483     * -1
1484     * Lc '\\@@_end_line:'
1485   )
1486 local languages = { }
1487 languages['python'] = python

```

6.3.3 The LPEG ocaml

```

1488 local Delim = Q ( P "[" + P "]" + S "[]" )
1489 local Punct = Q ( S ",:;!)" )
1490 local identifier =
1491   ( R "az" + R "AZ" + P "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
1492
1493 local Identifier = K ( 'Identifier' , identifier )
1494
1495 local Operator =
1496   K ( 'Operator' ,
1497     P "!=" + P "<>" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P "!="
1498     + P "||" + P "&&" + P "://" + P "***" + P ";" + P "::" + P "->"
1499     + P "+." + P "-." + P "*." + P "/"
1500     + S "--+/*%=<>&@|"
1501   )
1502
1503 local OperatorWord =
1504   K ( 'Operator.Word' ,
1505     P "and" + P "asr" + P "land" + P "lor" + P "lsl" + P "lxor"
1506     + P "mod" + P "or" )
1507
1508 local Keyword =
1509   K ( 'Keyword' ,
1510     P "as" + P "assert" + P "begin" + P "class" + P "constraint" + P "done"
1511     + P "do" + P "downto" + P "else" + P "end" + P "exception" + P "external"
1512     + P "false" + P "for" + P "function" + P "fun" + P "functor" + P "if"
1513     + P "in" + P "include" + P "inherit" + P "initializer" + P "lazy" + P "let"
1514     + P "match" + P "method" + P "module" + P "mutable" + P "new" + P "object"
1515     + P "of" + P "open" + P "private" + P "raise" + P "rec" + P "sig"
1516     + P "struct" + P "then" + P "to" + P "true" + P "try" + P "type"
1517     + P "value" + P "val" + P "virtual" + P "when" + P "while" + P "with" )
1518   + K ( 'Keyword.Constant' , P "true" + P "false" )
1519
1520
1521 local Builtin =
1522   K ( 'Name.Builtin' ,
1523     P "not" + P "incr" + P "decr" + P "fst" + P "snd"

```

²⁷Remember 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:`

```

1524     + P "String.length"
1525     + P "List.tl" + P "List.hd" + P "List.mem" + P "List.exists"
1526     + P "List.for_all" + P "List.filter" + P "List.length" + P "List.map"
1527     + P "List.iter"
1528     + P "Array.length" + P "Array.make" + P "Array.make_matrix"
1529     + P "Array.init" + P "Array.copy" + P "Array.copy" + P "Array.mem"
1530     + P "Array.exists" + P "Array.for_all" + P "Array.map" + P "Array.iter"
1531     + P "Queue.create" + P "Queue.is_empty" + P "Queue.push" + P "Queue.pop"
1532     + P "Stack.create" + P "Stack.is_empty" + P "Stack.push" + P "Stack.pop"
1533     + P "Hashtbl.create" + P "Hashtbl.add" + P "Hashtbl.remove"
1534     + P "Hashtbl.mem" + P "Hashtbl.find" + P "Hashtbl.find_opt"
1535     + P "Hashtbl.iter" )

```

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

```

1536 local Exception =
1537   K ( 'Exception' ,
1538       P "Division_by_zero" + P "End_of_File" + P "Failure"
1539     + P "Invalid_argument" + P "Match_failure" + P "Not_found"
1540     + P "Out_of_memory" + P "Stack_overflow" + P "Sys_blocked_io"
1541     + P "Sys_error" + P "Undefined_recursive_module" )

```

The characters in OCaml

```

1542 local Char =
1543   K ( 'String.Short' , P "'" * ( ( 1 - P "'" ) ^ 0 + P "\\'" ) * P "'" )

```

Beamer

```

1544 local BalancedBraces =
1545   P { "E" ,
1546       E =
1547         (
1548           P "{" * V "E" * P "}"
1549         +
1550           P "\" * ( 1 - S "\" ) ^ 0 * P "\" -- OCaml strings
1551         +
1552           ( 1 - S "{" )
1553         ) ^ 0
1554   }
1555 if piton_beamer
1556 then
1557   Beamer =
1558     L ( P "\\pause" * ( P "[" * ( 1 - P "]" ) ^ 0 * P "]" ) ^ -1 )
1559   +
1560     ( P "\\uncover" * Lc ( '\\@@_beamer_command:n{uncover}' )
1561   + P "\\only" * Lc ( '\\@@_beamer_command:n{only}' )
1562   + P "\\alert" * Lc ( '\\@@_beamer_command:n{alert}' )
1563   + P "\\visible" * Lc ( '\\@@_beamer_command:n{visible}' )
1564   + P "\\invisible" * Lc ( '\\@@_beamer_command:n{invisible}' )
1565   + P "\\action" * Lc ( '\\@@_beamer_command:n{action}' )
1566   )
1567   *
1568   L ( ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1 * P "{" )
1569   * K ( 'ParseAgain.noCR' , BalancedBraces )
1570   * L ( P "}" )
1571   +
1572   L (
1573     ( P "\\alt" )
1574     * P "<" * ( 1 - P ">" ) ^ 0 * P ">"
1575     * P "{"
1576   )
1577   * K ( 'ParseAgain.noCR' , BalancedBraces )

```

```

1578     * L ( P "{" )
1579     * K ( 'ParseAgain.noCR' , BalancedBraces )
1580     * L ( P "}" )
1581   +
1582   L (
1583     ( P "\\temporal" )
1584     * P "<" * ( 1 - P ">" ) ^ 0 * P ">"
1585     * P "{"
1586   )
1587   * K ( 'ParseAgain.noCR' , BalancedBraces )
1588   * L ( P "{" )
1589   * K ( 'ParseAgain.noCR' , BalancedBraces )
1590   * L ( P "{" )
1591   * K ( 'ParseAgain.noCR' , BalancedBraces )
1592   * L ( P "}" )
1593   BeamerBeginEnvironments =
1594     ( space ^ 0 *
1595       L (
1596         (
1597           P "\\begin{" * BeamerNamesEnvironments * "}"
1598           * ( P "<" * ( 1 - P ">" ) ^ 0 * P ">" ) ^ -1
1599         )
1600         * P "\r"
1601       ) ^ 0
1602   BeamerEndEnvironments =
1603     ( space ^ 0 *
1604       L ( P "\\end{" * BeamerNamesEnvironments * P "}" )
1605       * P "\r"
1606     ) ^ 0
1607   end

```

EOL

```

1608 local EOL =
1609   P "\r"
1610   *
1611   (
1612     ( space^0 * -1 )
1613     +
1614     Ct (
1615       Cc "EOL"
1616       *
1617       Ct (
1618         Lc "\\@@_end_line:"
1619         * BeamerEndEnvironments
1620         * BeamerBeginEnvironments
1621         * PromptHastyDetection
1622         * Lc "\\@@_newline: \\@@_begin_line:"
1623         * Prompt
1624       )
1625     )
1626   )
1627   *
1628   SpaceIndentation ^ 0
1629 %
1630 % \paragraph{The strings}
1631 %
1632 % We need a pattern |string| without captures because it will be used within the
1633 % comments of OCaml.
1634 % \begin{macrocode}
1635 local string =
1636   Q ( P "\"" )

```

```

1637     * (
1638         VisualSpace
1639         +
1640         Q ( ( 1 - S " \r" ) ^ 1 )
1641         +
1642         EOL
1643     ) ^ 0
1644     * Q ( P "\"" )
1645 local String = WithStyle ( 'String.Long' , 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 programmation, see the paragraphe *Lua’s long strings* in www.inf.puc-rio.br/~roberto/lpeg.

```

1646 local ext = ( R "az" + P "_" ) ^ 0
1647 local open = "{" * Cg(ext, 'init') * "|"
1648 local close = "|" * C(ext) * "}"
1649 local closeeq =
1650     Cmt ( close * Cb('init'),
1651         function (s, i, a, b) return a==b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

1652 local QuotedStringBis =
1653     WithStyle ( 'String.Long' ,
1654         (
1655             VisualSpace
1656             +
1657             Q ( ( 1 - S " \r" ) ^ 1 )
1658             +
1659             EOL
1660         ) ^ 0 )
1661

```

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.

```

1662 local QuotedString =
1663     C ( open * ( 1 - closeeq ) ^ 0 * close ) /
1664     ( 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).

```

1665 local Comment =
1666     WithStyle ( 'Comment' ,
1667         P {
1668             "A" ,
1669             A = Q "(" *
1670                 * ( V "A"
1671                     + Q ( ( 1 - P "(" - P ")" - S "\r$" ) ^ 1 ) -- $
1672                     + string
1673                     + P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$" -- $
1674                     + EOL
1675                 ) ^ 0
1676                 * Q ")"
1677         } )

```

The DefFunction

```
1678 local DefFunction =
1679   K ( 'Keyword' , P "let rec" + P "let" + P "and" )
1680   * Space
1681   * K ( 'Name.Function.Internal' , identifier )
1682   * Space
1683   * # ( P "=" * space * P "function" + ( 1 - P "=" ) )
```

The parameters of the types

```
1684 local TypeParameter = K ( 'TypeParameter' , P "'" * alpha * # ( 1 - P "'" ) )
```

The main LPEG

 First, the main loop :

```
1685 MainLoopOCaml =
1686   ( ( space1 * -1 )
1687     + EOL
1688     + Space
1689     + Tab
1690     + Escape
1691     + Beamer
1692     + TypeParameter
1693     + String + QuotedString + Char
1694     + Comment
1695     + Delim
1696     + Operator
1697     + Punct
1698     + FromImport
1699     + ImportAs
1700     + Exception
1701     + DefFunction
1702     + Keyword * ( Space + Punct + Delim + EOL + -1 )
1703     + OperatorWord * ( Space + Punct + Delim + EOL + -1 )
1704     + Builtin * ( Space + Punct + Delim + EOL + -1 )
1705     + Identifier
1706     + Number
1707     + Word
1708   ) ^ 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:`²⁸.

```
1709 local ocaml = P ( true )
1710
1711 ocaml =
1712   Ct (
1713     ( ( space - P "\r" ) ^ 0 * P "\r" ) ^ -1
1714     * BeamerBeginEnvironments
1715     * Lc ( '\\@@_begin_line:' )
1716     * SpaceIndentation ^ 0
1717     * MainLoopOCaml
1718     * -1
1719     * Lc ( '\\@@_end_line:' )
1720   )
1721 languages['ocaml'] = ocaml
```

²⁸Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

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.

```
1722 function piton.Parse(language,code)
1723   local t = languages[language] : match ( code )
1724   local left_stack = {}
1725   local right_stack = {}
1726   for _ , one_item in ipairs(t)
1727   do
1728     if one_item[1] == "EOL"
1729     then
1730       for _ , s in ipairs(right_stack)
1731       do tex.sprint( s )
1732       end
1733       for _ , s in ipairs(one_item[2])
1734       do tex.tprint(s)
1735       end
1736       for _ , s in ipairs(left_stack)
1737       do tex.sprint( s )
1738       end
1739     else
1740       if one_item[1] == "Open"
1741       then
1742         tex.sprint( one_item[2] )
1743         table.insert(left_stack,one_item[2])
1744         table.insert(right_stack,one_item[3])
1745       else
1746         if one_item[1] == "Close"
1747         then
1748           tex.sprint( right_stack[#right_stack] )
1749           left_stack[#left_stack] = nil
1750           right_stack[#right_stack] = nil
1751         else
1752           tex.tprint(one_item)
1753         end
1754       end
1755     end
1756   end
1757 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.

```
1758 function piton.ParseFile(language,name,first_line,last_line)
1759   s = ''
1760   local i = 0
1761   for line in io.lines(name)
1762   do i = i + 1
1763     if i >= first_line
1764     then s = s .. '\r' .. line
1765     end
1766     if i >= last_line then break end
1767   end
1768   piton.Parse(language,s)
1769 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 `#`.

```

1770 function piton.ParseBis(language,code)
1771   local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
1772   return piton.Parse(language,s)
1773 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.

```

1774 function piton.ParseTer(language,code)
1775   local s = ( Cs ( ( P '\\@@_breakable_space:' / ' ' + 1 ) ^ 0 ) )
1776             : match ( code )
1777   return piton.Parse(language,s)
1778 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 .

```

1779 local function gobble(n,code)
1780   function concat(acc,new_value)
1781     return acc .. new_value
1782   end
1783   if n==0
1784   then return code
1785   else
1786     return Cf (
1787       Cc ( "" ) *
1788       ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
1789       * ( C ( P "\r" )
1790         * ( 1 - P "\r" ) ^ (-n)
1791         * C ( ( 1 - P "\r" ) ^ 0 )
1792       ) ^ 0 ,
1793       concat
1794     ) : match ( code )
1795   end
1796 end

```

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

```

1797 local function add(acc,new_value)
1798   return acc + new_value
1799 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.

```

1800 local AutoGobbleLPEG =
1801   ( space ^ 0 * P "\r" ) ^ -1
1802   * Cf (
1803     (

```

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

```

1804      ( P " " ) ^ 0 * P "\r"
1805      +
1806      Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1807      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * P "\r"
1808      ) ^ 0

```

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

```

1809      *
1810      ( Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1811      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,
1812      math.min
1813      )

```

The following LPEG is similar but works with the indentations.

```

1814 local TabsAutoGobbleLPEG =
1815   ( space ^ 0 * P "\r" ) ^ -1
1816   * Cf (
1817     (
1818       ( P "\t" ) ^ 0 * P "\r"
1819       +
1820       Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )
1821       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * P "\r"
1822     ) ^ 0
1823     *
1824     ( Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )
1825     * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,
1826     math.min
1827   )

```

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.

```

1828 local EnvGobbleLPEG =
1829   ( ( 1 - P "\r" ) ^ 0 * P "\r" ) ^ 0
1830   * Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add ) * -1

```

```

1831 function piton.GobbleParse(language,n,code)
1832   if n==-1
1833     then n = AutoGobbleLPEG : match(code)
1834   else if n==-2
1835     then n = EnvGobbleLPEG : match(code)
1836   else if n==-3
1837     then n = TabsAutoGobbleLPEG : match(code)
1838   end
1839 end
1840 end
1841 piton.Parse(language,gobble(n,code))
1842 end

```

6.3.7 To count the number of lines

```

1843 function piton.CountLines(code)
1844   local count = 0
1845   for i in code : gmatch ( "\r" ) do count = count + 1 end
1846   tex.sprint(
1847     luatexbase.catcodetables.expl ,
1848     '\\int_set:Nn \\l_@@_nb_lines_int {' .. count .. '}' )

```

```

1849 end

1850 function piton.CountNonEmptyLines(code)
1851   local count = 0
1852   count =
1853   ( Cf ( Cc(0) *
1854     (
1855       ( P " " ) ^ 0 * P "\r"
1856       + ( 1 - P "\r" ) ^ 0 * P "\r" * Cc(1)
1857     ) ^ 0
1858     * ( 1 - P "\r" ) ^ 0 ,
1859     add
1860   ) * -1 ) : match (code)
1861   tex.sprint(
1862     luatexbase.catcodetables.expl ,
1863     '\\int_set:Nn \\l_@@_nb_non_empty_lines_int {' .. count .. '}' )
1864 end

1865 function piton.CountLinesFile(name)
1866   local count = 0
1867   for line in io.lines(name) do count = count + 1 end
1868   tex.sprint(
1869     luatexbase.catcodetables.expl ,
1870     '\\int_set:Nn \\l_@@_nb_lines_int {' .. count .. '}' )
1871 end

1872 function piton.CountNonEmptyLinesFile(name)
1873   local count = 0
1874   for line in io.lines(name)
1875   do if not ( ( ( P " " ) ^ 0 * -1 ) : match ( line ) )
1876     then count = count + 1
1877     end
1878   end
1879   tex.sprint(
1880     luatexbase.catcodetables.expl ,
1881     '\\int_set:Nn \\l_@@_nb_non_empty_lines_int {' .. count .. '}' )
1882 end
1883 \end{luacode*}

```

7 History

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 environments `{uncoverenv}`, `{onlyenv}`, `{visibleenv}` and `{invisibleenv}`

Changes between versions 1.0 and 1.1

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

Changes between versions 0.99 and 1.0

New key `tabs-auto-gobble`.

Changes between versions 0.95 and 0.99

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

Changes between versions 0.9 and 0.95

New key `show-spaces`.

The key `left-margin` now accepts the special value `auto`.

New key `latex-comment` at load-time and replacement of `##` by `#>`

New key `math-comments` at load-time.

New keys `first-line` and `last-line` for the command `\InputPitonFile`.

Changes between versions 0.8 and 0.9

New key `tab-size`.

Integer value for the key `splittable`.

Changes between versions 0.7 and 0.8

New keys `footnote` and `footnotehyper` at load-time.

New key `left-margin`.

Changes between versions 0.6 and 0.7

New keys `resume`, `splittable` and `background-color` in `\PitonOptions`.

The file `piton.lua` has been embedded in the file `piton.sty`. That means that the extension `piton` is now entirely contained in the file `piton.sty`.

Contents

1	Presentation	1
2	Use of the package	2
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
3	Customization	3
3.1	The command <code>\PitonOptions</code>	3
3.2	The styles	5
3.3	Creation of new environments	6

4	Advanced features	6
4.1	Highlighting some identifiers	6
4.2	Mechanisms to escape to LaTeX	8
4.2.1	The “LaTeX comments”	8
4.2.2	The key “math-comments”	8
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”	10
4.3.2	Commands of Beamer allowed in {Piton} and \PitonInputFile	10
4.3.3	Environments of Beamer allowed in {Piton} and \PitonInputFile	11
4.4	Page breaks and line breaks	12
4.4.1	Page breaks	12
4.4.2	Line breaks	12
4.5	Footnotes in the environments of piton	13
4.6	Tabulations	13
5	Examples	13
5.1	Line numbering	13
5.2	Formatting of the LaTeX comments	14
5.3	Notes in the listings	15
5.4	An example of tuning of the styles	16
5.5	Use with pyluatex	17
6	Implementation	20
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	27
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	36
6.2.9	The initial style	38
6.2.10	Highlighting some identifiers	39
6.2.11	Security	40
6.2.12	The error messages of the package	40
6.3	The Lua part of the implementation	41
6.3.1	Special functions dealing with LPEG	42
6.3.2	The LPEG python	44
6.3.3	The LPEG ocaml	54
6.3.4	The function Parse	59
6.3.5	Two variants of the function Parse with integrated preprocessors	60
6.3.6	Preprocessors of the function Parse for gobble	60
6.3.7	To count the number of lines	61
7	History	62