The package piton[*]

F. Pantigny
fpantigny@wanadoo.fr

April 25, 2023

Abstract

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

1 Presentation

The package piton uses the Lua library LPEG[1] 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}`.

```python
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) = π/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 1.5x of piton, at the date of 2023/04/25.

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

[2]This LaTeX escape has been done by beginning the comment by `#>`.

2 Use of the package

2.1 Loading the package

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

- the package piton uses the package xcolor (but piton does *not* load xcolor: if xcolor is not loaded before the `\begin{document}`, a fatal error will be raised).

- the package piton must be used with LuaLaTeX exclusively: if another LaTeX engine (`latex`, `pdflatex`, `xelatex`,…) is used, a fatal error will be raised.

2.2 The tools provided to the user

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

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

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

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

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

- The command `\PitonInputFile` is used to insert and typeset a whole external file.

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

2.3 The syntax of the command \piton

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

- Syntax `\piton{...}`

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

  - several consecutive spaces will be replaced by only one space,
    but the command `\␣` is provided to force the insertion of a space;
  - it's not possible to use `%` inside the argument,
    but the command `\%` is provided to insert a `%`;
  - the braces must be appear by pairs correctly nested
    but the commands `\{` and `\}` are also provided for individual braces;
  - the LaTeX commands[3] are fully expanded and not executed,
    so it's possible to use `\\` to insert a backslash.

---

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

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

Examples :

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

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 :

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

3 Customization

3.1 The command \PitonOptions

The command `\PitonOptions` takes in as argument a comma-separated list of *key=value* pairs. The scope of the settings done by that command is the current TeX group.[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` of `all-line-numbers`) and the beginning of the lines of code. The initial value is 0.7 em.

---

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

---

[6] The package piton simply uses the current monospaced font. The best way to change that font is to use the command `\setmonofont` of the package fontspec.

[7] cf. 4.4.2 p. 12

```
1   from math import pi

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

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

3.2   The styles

The package piton provides the command `\SetPitonStyle` to customize the different styles used to format the syntactic elements of the Python listings. The customizations done by that command are limited to the current TeX group.[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*.

---

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

[9]See: https://pygments.org/styles/. Remark that, by default, Pygments provides for its style manni a colored background whose color is the HTML color #F0F3F3. It's possible to have the same color in {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}{O{}}{\PitonOptions{#1}}{}`

If one wishes to format Python code in a box of **tcolorbox**, it's possible to define an environment `{Python}` with the following code (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 formating for some identifiers with the key `identifiers` of `\PitonOptions`.
That key takes in as argument a value of the following format:
  `{ names = names, style = instructions }`

- *names* is a (comma-separated) list of identifiers names;

- *instructions* is a list of LaTeX instructions of the same type that `piton` "styles" previously presented (cf 3.2 p. 5).

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

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

\begin{Piton}
def tri(l):
    """Segmentation sort"""
    if len(l) <= 1:
        return l
    else:
        a = l[0]
        l1 = [ x for x in l[1:] if x < a  ]
        l2 = [ x for x in l[1:] if x >= a ]
        return tri(l1) + [a] + tri(l2)
\end{Piton}
```

```python
def tri(l):
    """Segmentation sort"""
    if len(l) <= 1:
        return l
    else:
        a = l[0]
        l1 = [ x for x in l[1:] if x < a  ]
        l2 = [ x for x in l[1:] if x >= a ]
        return tri(l1) + [a] + tri(l2)
```

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

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

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

```python
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 aslo 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`.

---

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

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

```
def square(x):
    return x*x # compute x²
```

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{docuemnt}`).

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 programmation of the mathematical factorial, we decide to highlight in yellow the instruction which contains the recursive call. That example uses the command `\highLight` of lua-ul (that package requires itself the package luacolor).

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

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

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

```
\makeatletter
\let\Yellow\@highLight
\makeatother
```

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

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

Caution : The escape to LaTeX allowed by the 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 strings[14] of Python are not considered.

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

Here is a complete example of file:

---

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

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

[13]One should remark that it's also possible to use the command `\pause` in a "LaTeX comment", that is to say by writing `#> \pause`. By this way, if the Python code is copied, it's still executable by Python

[14]The short strings of Python are the strings delimited by characters `'` or the characters `"` and not `'''` nor `"""`. In Python, the short strings can't extend on several lines.

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

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

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):
    """Compure 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
hightlighted.

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 conjonction 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 coresponds 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}
```

---

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

```
    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) ant 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).

```
\PitonOptions{background-color=gray!10, left-margin = auto, line-numbers}
\begin{Piton}
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{Piton}
```

```
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 `\SetPitonStyle`) in order to display the LaTeX comments (which begin with `#>`) aligned on the right margin.

```
\PitonOptions{background-color=gray!10}
\SetPitonStyle{Comment.LaTeX = \hfill \normalfont\color{gray}}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)        #> 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{Piton}
```

```
 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.

```
\PitonOptions{background-color=gray!10}
\NewDocumentCommand{\MyLaTeXCommand}{m}{\hfill \normalfont\itshape\rlap{\quad #1}}
\SetPitonStyle{Comment.LaTeX = \MyLaTeXCommand}
\begin{minipage}{12cm}
\begin{Piton}
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)                          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
```

5.3   Notes in the listings

In order to be able to extract the notes (which are typeset with the command \footnote), the
extension piton must be loaded with the key footnote or the key footenotehyper as explained in the
section 4.5 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:
```

---

[16] First recursive call.
[17] Second 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 *DejaVu 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}
```

---

[18]First recursive call.
[19]Second recursive call.
[20]See: 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}
  }
```

```python
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) = π/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())
```

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

6 Implementation

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

6.1 Introduction

The main job of the package `piton` is to take in as input a Python listing and to send back to LaTeX as output that code *with interlaced LaTeX instructions of formatting.*
In fact, all that job is done by a LPEG called `python`. That LPEG, when matched against the string of a Python listing, returns as capture a Lua table containing data to send to LaTeX. The only thing to do after will be to apply `tex.tprint` to each element of that table.[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).

---

[21]Recall that `tex.tprint` takes in as argument a Lua table whose first component is a "catcode table" and the second element a string. The string will be sent to LaTeX with the regime of catcodes specified by the catcode table. If no catcode table is provided, the standard catcodes of LaTeX will be used.

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

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

6.2 The L3 part of the implementation

6.2.1 Declaration of the package

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

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

17 \RequirePackage { luatexbase }
```

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

The boolean \c_@@_footnote_bool will indicate if the option footnote is used, but quicky, it will also be set to true if the option footnotehyper is used.
```
19 \bool_new:N \c_@@_footnote_bool
```

The following boolean corresponds to the key math-comments (only at load-time).
```
20 \bool_new:N \c_@@_math_comments_bool
```

The following boolean corresponds to the key beamer.
```
21 \bool_new:N \c_@@_beamer_bool
```

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

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

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

```
44  \ProcessKeysOptions { piton / package }

45  \begingroup
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       {
185         \tl_if_empty:NF \l_@@_prompt_bg_color_tl % added 2023-04-24
186           { \clist_set:NV \l_@@_bg_color_clist \l_@@_prompt_bg_color_tl }
187       }
188   }
```

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

```
189 \clist_new:N \g_@@_current_style_clist
190 \clist_set:Nn \g_@@_current_style_clist { __end }
```

The element __end is an arbitrary syntactic marker.

```
191 \cs_new_protected:Npn \@@_close_current_styles:
192   {
193     \int_set:Nn \l_tmpa_int { \clist_count:N \g_@@_current_style_clist - 1  }
194     \exp_args:NV \@@_close_n_styles:n \l_tmpa_int
195   }
196 \cs_new_protected:Npn \@@_close_n_styles:n #1
197   {
198     \int_compare:nNnT { #1 } > 0
199       {
200         \@@_close_brace:
201         \@@_close_brace:
202         \@@_close_n_styles:n { #1 - 1 }
203       }
204   }
205 \cs_new_protected:Npn \@@_open_current_styles:
206   { \exp_last_unbraced:NV \@@_open_styles:w \g_@@_current_style_clist , }
207 \cs_new_protected:Npn \@@_open_styles:w #1 ,
208   {
209     \tl_if_eq:nnF { #1 } { __end }
210       { \@@_open_brace: #1 \@@_open_brace: \@@_open_styles:w }
211   }
212 \cs_new_protected:Npn \@@_pop_style:
213   {
214     \clist_greverse:N \g_@@_current_style_clist
215     \clist_gpop:NN \g_@@_current_style_clist \l_tmpa_tl
216     \clist_gpop:NN \g_@@_current_style_clist \l_tmpa_tl
217     \clist_gpush:Nn \g_@@_current_style_clist { __end }
218     \clist_greverse:N \g_@@_current_style_clist
219   }
220 \cs_new_protected:Npn \@@_push_style:n #1
221   {
222     \clist_greverse:N \g_@@_current_style_clist
223     \clist_gpop:NN \g_@@_current_style_clist \l_tmpa_tl
224     \clist_gpush:Nn \g_@@_current_style_clist { #1 }
225     \clist_gpush:Nn \g_@@_current_style_clist { __end }
226     \clist_greverse:N \g_@@_current_style_clist
227   }
228 \cs_new_protected:Npn \@@_push_and_exec:n #1
229   {
230     \@@_push_style:n { #1 }
231     \@@_open_brace: #1 \@@_open_brace:
232   }
```

6.2.3 Treatment of a line of code

```
233 \cs_new_protected:Npn \@@_replace_spaces:n #1
234   {
235     \tl_set:Nn \l_tmpa_tl { #1 }
236     \bool_if:NTF \l_@@_show_spaces_bool
237       { \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl } % U+2423
238         {
```

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.

```
239         \bool_if:NT \l_@@_break_lines_in_Piton_bool
240           {
241             \regex_replace_all:nnN
242               { \x20 }
243               { \c { @@_breakable_space: } }
244               \l_tmpa_tl
245           }
246         }
247     \l_tmpa_tl
248   }
249 \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.

```
250 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
251   {
252     \group_begin:
253     \g_@@_begin_line_hook_tl
254     \int_gzero:N \g_@@_indentation_int
```

Be careful: there is curryfication in the following lines.

```
255     \bool_if:NTF \l_@@_slim_bool
256       { \hcoffin_set:Nn \l_tmpa_coffin }
257       {
258         \clist_if_empty:NTF \l_@@_bg_color_clist
259           {
260             \vcoffin_set:Nnn \l_tmpa_coffin
261               { \dim_eval:n { \linewidth - \l_@@_left_margin_dim } }
262           }
263           {
264             \vcoffin_set:Nnn \l_tmpa_coffin
265               { \dim_eval:n { \linewidth - \l_@@_left_margin_dim - 0.5 em } }
266           }
267       }
268       {
269         \language = -1
270         \raggedright
271         \strut
272         \@@_replace_spaces:n { #1 }
273         \strut \hfil
274       }
275     \hbox_set:Nn \l_tmpa_box
276       {
277         \skip_horizontal:N \l_@@_left_margin_dim
278         \bool_if:NT \l_@@_line_numbers_bool
279           {
280             \bool_if:NF \l_@@_all_line_numbers_bool
281               { \tl_if_eq:nnF  { #1 } { \PitonStyle {Prompt}{} } }
282             \@@_print_number:
283           }
```

```
284        \clist_if_empty:NF \l_@@_bg_color_clist
285          {
286            \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
287              {
288                \bool_if:NF \l_@@_left_margin_auto_bool
289                  { \skip_horizontal:n { 0.5 em } }
290              }
291          }
292        \coffin_typeset:Nnnnn \l_tmpa_coffin T l \c_zero_dim \c_zero_dim
293      }
```

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

```
294      \dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_width_dim
295        { \dim_gset:Nn \g_@@_width_dim { \box_wd:N \l_tmpa_box } }
296      \box_set_dp:Nn \l_tmpa_box { \box_dp:N \l_tmpa_box + 1.25 pt }
297      \box_set_ht:Nn \l_tmpa_box { \box_ht:N \l_tmpa_box + 1.25 pt }
298      \clist_if_empty:NTF \l_@@_bg_color_clist
299        { \box_use_drop:N \l_tmpa_box }
300        {
301          \vbox_top:n
302            {
303              \hbox:n
304                {
305                  \@@_color:N \l_@@_bg_color_clist
306                  \vrule height \box_ht:N \l_tmpa_box
307                          depth \box_dp:N \l_tmpa_box
308                          width \l_@@_width_on_aux_dim
309                }
310              \skip_vertical:n { - \box_ht_plus_dp:N \l_tmpa_box }
311              \box_set_wd:Nn \l_tmpa_box \l_@@_width_on_aux_dim
312              \box_use_drop:N \l_tmpa_box
313            }
314        }
315      \vspace { - 2.5 pt }
316      \group_end:
317      \tl_gclear:N \g_@@_begin_line_hook_tl
318    }
```

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).

```
319  \cs_set_protected:Npn \@@_color:N #1
320    {
321      \int_set:Nn \l_tmpa_int { \clist_count:N #1 }
322      \int_set:Nn \l_tmpb_int { \int_mod:nn \g_@@_line_int \l_tmpa_int + 1 }
323      \tl_set:Nx \l_tmpa_tl { \clist_item:Nn #1 \l_tmpb_int }
324      \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).

```
325        { \dim_zero:N \l_@@_width_on_aux_dim }
326        { \exp_args:NV \@@_color_i:n \l_tmpa_tl }
327    }
```

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} }`.

```
328  \cs_set_protected:Npn \@@_color_i:n #1
329    {
330      \tl_if_head_eq_meaning:nNTF { #1 } [
331        {
332          \tl_set:Nn \l_tmpa_tl { #1 }
333          \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
334          \exp_last_unbraced:NV \color \l_tmpa_tl
335        }
336        { \color { #1 } }
```

```
337      }
338  \cs_generate_variant:Nn \@@_color:n { V }

339  \cs_new_protected:Npn \@@_newline:
340    {
341      \int_gincr:N \g_@@_line_int
342      \int_compare:nNnT \g_@@_line_int > { \l_@@_splittable_int - 1 }
343        {
344          \int_compare:nNnT
345            { \l_@@_nb_lines_int - \g_@@_line_int } > \l_@@_splittable_int
346            {
347              \egroup
348              \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
349              \newline
350              \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
351              \vtop \bgroup
352            }
353        }
354    }

355  \cs_set_protected:Npn \@@_breakable_space:
356    {
357      \discretionary
358        { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
359        {
360          \hbox_overlap_left:n
361            {
362              {
363                \normalfont \footnotesize \color { gray }
364                \l_@@_continuation_symbol_tl
365              }
366              \skip_horizontal:n { 0.3 em }
367              \clist_if_empty:NF \l_@@_bg_color_clist
368                { \skip_horizontal:n { 0.5 em } }
369            }
370          \bool_if:NT \l_@@_indent_broken_lines_bool
371            {
372              \hbox:n
373                {
374                  \prg_replicate:nn { \g_@@_indentation_int } { ~ }
375                  { \color { gray } \l_@@_csoi_tl }
376                }
377            }
378        }
379        { \hbox { ~ } }
380    }
```

6.2.4 PitonOptions

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

```
381  \bool_new:N \l_@@_line_numbers_bool
382  \bool_new:N \l_@@_all_line_numbers_bool
```

The following flag corresponds to the key `resume`.

```
383  \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.

```
384  \keys_define:nn { PitonOptions }
385    {
386      language          .str_set:N          = \l_@@_language_str ,
```

```
387    language         .value_required:n  = true ,
388    gobble           .int_set:N         = \l_@@_gobble_int ,
389    gobble           .value_required:n  = true ,
390    auto-gobble      .code:n            = \int_set:Nn \l_@@_gobble_int { -1 } ,
391    auto-gobble      .value_forbidden:n = true ,
392    env-gobble       .code:n            = \int_set:Nn \l_@@_gobble_int { -2 } ,
393    env-gobble       .value_forbidden:n = true ,
394    tabs-auto-gobble .code:n            = \int_set:Nn \l_@@_gobble_int { -3 } ,
395    tabs-auto-gobble .value_forbidden:n = true ,
396    line-numbers     .bool_set:N        = \l_@@_line_numbers_bool ,
397    line-numbers     .default:n         = true ,
398    all-line-numbers .code:n =
399      \bool_set_true:N \l_@@_line_numbers_bool
400      \bool_set_true:N \l_@@_all_line_numbers_bool ,
401    all-line-numbers .value_forbidden:n = true  ,
402    resume           .bool_set:N        = \l_@@_resume_bool ,
403    resume           .value_forbidden:n = true ,
404    splittable       .int_set:N         = \l_@@_splittable_int ,
405    splittable       .default:n         = 1 ,
406    background-color .clist_set:N       = \l_@@_bg_color_clist ,
407    background-color .value_required:n  = true ,
408    prompt-background-color .tl_set:N        = \l_@@_prompt_bg_color_tl ,
409    prompt-background-color .value_required:n = true ,
410    slim             .bool_set:N        = \l_@@_slim_bool ,
411    slim             .default:n         = true ,
412    left-margin      .code:n =
413      \str_if_eq:nnTF { #1 } { auto }
414        {
415          \dim_zero:N \l_@@_left_margin_dim
416          \bool_set_true:N \l_@@_left_margin_auto_bool
417        }
418        { \dim_set:Nn \l_@@_left_margin_dim { #1 } } ,
419    left-margin      .value_required:n  = true ,
420    numbers-sep      .dim_set:N         = \l_@@_numbers_sep_dim ,
421    numbers-sep      .value_required:n  = true ,
422    tab-size         .code:n            = \@@_set_tab_tl:n { #1 } ,
423    tab-size         .value_required:n  = true ,
424    show-spaces      .bool_set:N        = \l_@@_show_spaces_bool ,
425    show-spaces      .default:n         = true ,
426    show-spaces-in-strings .code:n      = \tl_set:Nn \l_@@_space_tl { ␣ } , % U+2423
427    show-spaces-in-strings .value_forbidden:n = true ,
428    break-lines-in-Piton .bool_set:N    = \l_@@_break_lines_in_Piton_bool ,
429    break-lines-in-Piton .default:n     = true ,
430    break-lines-in-piton .bool_set:N    = \l_@@_break_lines_in_piton_bool ,
431    break-lines-in-piton .default:n     = true ,
432    break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
433    break-lines .value_forbidden:n      = true ,
434    indent-broken-lines .bool_set:N     = \l_@@_indent_broken_lines_bool ,
435    indent-broken-lines .default:n      = true ,
436    end-of-broken-line  .tl_set:N       = \l_@@_end_of_broken_line_tl ,
437    end-of-broken-line  .value_required:n = true ,
438    continuation-symbol .tl_set:N       = \l_@@_continuation_symbol_tl ,
439    continuation-symbol .value_required:n = true ,
440    continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
441    continuation-symbol-on-indentation .value_required:n = true ,
442    unknown          .code:n =
443      \msg_error:nn { piton } { Unknown~key~for~PitonOptions }
444  }
```

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

```
445 \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`).

```
446  \int_new:N \g_@@_visual_line_int

447  \cs_new_protected:Npn \@@_print_number:
448    {
449      \int_gincr:N \g_@@_visual_line_int
450      \hbox_overlap_left:n
451        {
452          { \color { gray } \footnotesize \int_to_arabic:n \g_@@_visual_line_int }
453          \skip_horizontal:N \l_@@_numbers_sep_dim
454        }
455    }
```

6.2.6 The command to write on the aux file

```
456  \cs_new_protected:Npn \@@_write_aux:
457    {
458      \tl_if_empty:NF \g_@@_aux_tl
459        {
460          \iow_now:Nn \@mainaux { \ExplSyntaxOn }
461          \iow_now:Nx \@mainaux
462            {
463              \tl_gset:cn { c_@@_ \int_use:N \g_@@_env_int _ tl }
464                { \exp_not:V \g_@@_aux_tl }
465            }
466          \iow_now:Nn \@mainaux { \ExplSyntaxOff }
467        }
468      \tl_gclear:N \g_@@_aux_tl
469    }

470  \cs_new_protected:Npn \@@_width_to_aux:
471    {
472      \bool_if:NT \l_@@_slim_bool
473        {
474          \clist_if_empty:NF \l_@@_bg_color_clist
475            {
476              \tl_gput_right:Nx \g_@@_aux_tl
477                {
478                  \dim_set:Nn \l_@@_width_on_aux_dim
479                    { \dim_eval:n { \g_@@_width_dim + 0.5 em } } }
480                }
481            }
482        }
483    }
```

6.2.7 The main commands and environments for the final user

```
484  \NewDocumentCommand { \piton } { } 
485    { \peek_meaning:NTF \bgroup \@@_piton_standard \@@_piton_verbatim }
486  \NewDocumentCommand { \@@_piton_standard } { m }
487    {
488      \group_begin:
489      \ttfamily
```

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

```
490      \automatichyphenmode = 1
491      \cs_set_eq:NN \\ \c_backslash_str
492      \cs_set_eq:NN \% \c_percent_str
```

```
493    \cs_set_eq:NN \{ \c_left_brace_str
494    \cs_set_eq:NN \} \c_right_brace_str
495    \cs_set_eq:NN \$ \c_dollar_str
496    \cs_set_eq:cN { ~ } \space
497    \cs_set_protected:Npn \@@_begin_line: { }
498    \cs_set_protected:Npn \@@_end_line: { }
499    \tl_set:Nx \l_tmpa_tl
500      {
501        \lua_now:e
502          { piton.ParseBis('\l_@@_language_str',token.scan_string()) }
503          { #1 }
504      }
505    \bool_if:NTF \l_@@_show_spaces_bool
506      { \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.

```
507      {
508        \bool_if:NT \l_@@_break_lines_in_piton_bool
509          { \regex_replace_all:nnN { \x20 } { \x20 } \l_tmpa_tl }
510      }
511    \l_tmpa_tl
512    \group_end:
513    }
514 \NewDocumentCommand { \@@_piton_verbatim } { v }
515    {
516      \group_begin:
517      \ttfamily
518      \automatichyphenmode = 1
519      \cs_set_protected:Npn \@@_begin_line: { }
520      \cs_set_protected:Npn \@@_end_line: { }
521      \tl_set:Nx \l_tmpa_tl
522        {
523          \lua_now:e
524            { piton.Parse('\l_@@_language_str',token.scan_string()) }
525            { #1 }
526        }
527      \bool_if:NT \l_@@_show_spaces_bool
528        { \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl } % U+2423
529      \l_tmpa_tl
530      \group_end:
531    }
```

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).

```
532 \cs_new_protected:Npn \@@_piton:n #1
533    {
534      \group_begin:
535      \cs_set_protected:Npn \@@_begin_line: { }
536      \cs_set_protected:Npn \@@_end_line: { }
537      \bool_lazy_or:nnTF
538        \l_@@_break_lines_in_piton_bool
539        \l_@@_break_lines_in_Piton_bool
540        {
541          \tl_set:Nx \l_tmpa_tl
542            {
543              \lua_now:e
544                { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
545                { #1 }
546            }
547        }
```

```
548     {
549       \tl_set:Nx \l_tmpa_tl
550         {
551           \lua_now:e
552             { piton.Parse('\l_@@_language_str',token.scan_string()) }
553             { #1 }
554         }
555     }
556     \bool_if:NT \l_@@_show_spaces_bool
557       { \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl } % U+2423
558     \l_tmpa_tl
559     \group_end:
560   }
```

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

```
561 \cs_new_protected:Npn \@@_piton_no_cr:n #1
562   {
563     \group_begin:
564     \cs_set_protected:Npn \@@_begin_line: { }
565     \cs_set_protected:Npn \@@_end_line: { }
566     \cs_set_protected:Npn \@@_newline:
567       { \msg_fatal:nn { piton } { cr~not~allowed } }
568     \bool_lazy_or:nnTF
569       \l_@@_break_lines_in_piton_bool
570       \l_@@_break_lines_in_Piton_bool
571       {
572         \tl_set:Nx \l_tmpa_tl
573           {
574             \lua_now:e
575               { piton.ParseTer('\l_@@_language_str',token.scan_string()) }
576               { #1 }
577           }
578       }
579       {
580         \tl_set:Nx \l_tmpa_tl
581           {
582             \lua_now:e
583               { piton.Parse('\l_@@_language_str',token.scan_string()) }
584               { #1 }
585           }
586       }
587     \bool_if:NT \l_@@_show_spaces_bool
588       { \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl } % U+2423
589     \l_tmpa_tl
590     \group_end:
591   }
```

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

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

```
605    \dim_zero:N \parindent
606    \cs_set_eq:NN \label \@@_label:n
607  }


608  \keys_define:nn { PitonInputFile }
609    {
610      first-line .int_set:N = \l_@@_first_line_int ,
611      first-line .value_required:n = true ,
612      last-line .int_set:N = \l_@@_last_line_int ,
613      last-line .value_required:n = true ,
614    }


615  \NewDocumentCommand { \PitonInputFile } { d < > O { } m }
616    {
617      \tl_if_novalue:nF { #1 }
618        {
619          \bool_if:NTF \c_@@_beamer_bool
620            { \begin { uncoverenv } < #1 > }
621            { \msg_error:nn { piton } { overlay~without~beamer } }
622        }
623      \group_begin:
624        \int_zero_new:N \l_@@_first_line_int
625        \int_zero_new:N \l_@@_last_line_int
626        \int_set_eq:NN \l_@@_last_line_int \c_max_int
627        \keys_set:nn { PitonInputFile } { #2 }
628        \@@_pre_env:
629        \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.

```
630          \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.

```
631          \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
632            {
633              \hbox_set:Nn \l_tmpa_box
634                {
635                  \footnotesize
636                  \bool_if:NTF \l_@@_all_line_numbers_bool
637                    {
638                      \int_to_arabic:n
639                        { \g_@@_visual_line_int + \l_@@_nb_lines_int }
640                    }
641                    {
642                      \lua_now:n
643                        { piton.CountNonEmptyLinesFile(token.scan_argument()) }
644                        { #3 }
645                      \int_to_arabic:n
646                        { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
647                    }
648                }
649              \dim_set:Nn \l_@@_left_margin_dim
650                { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
651            }
```

Now, the main job.

```
652          \ttfamily
653          \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
654          \vtop \bgroup
655          \lua_now:e
656            {
657              piton.ParseFile('\l_@@_language_str',token.scan_argument() ,
```

```
658          \int_use:N \l_@@_first_line_int ,
659          \int_use:N \l_@@_last_line_int )
660        }
661        { #3 }
662      \egroup
663      \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
664      \@@_width_to_aux:
665    \group_end:
666    \tl_if_novalue:nF { #1 }
667      { \bool_if:NT \c_@@_beamer_bool { \end { uncoverenv } } }
668    \@@_write_aux:
669  }


670 \NewDocumentCommand { \NewPitonEnvironment } { m m m m }
671   {
```

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.

```
672      \use:x
673        {
674          \cs_set_protected:Npn
675            \use:c { _@@_collect_ #1 :w }
676            ####1
677            \c_backslash_str end \c_left_brace_str #1 \c_right_brace_str
678        }
679          {
680            \group_end:
681            \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.

```
682            \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.

```
683            \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
684              {
685                \bool_if:NTF \l_@@_all_line_numbers_bool
686                  {
687                    \hbox_set:Nn \l_tmpa_box
688                      {
689                        \footnotesize
690                        \int_to_arabic:n
691                          { \g_@@_visual_line_int + \l_@@_nb_lines_int }
692                      }
693                  }
694                  {
695                    \lua_now:n
696                      { piton.CountNonEmptyLines(token.scan_argument()) }
697                      { ##1 }
698                    \hbox_set:Nn \l_tmpa_box
699                      {
700                        \footnotesize
701                        \int_to_arabic:n
702                          { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
703                      }
704                  }
705                \dim_set:Nn \l_@@_left_margin_dim
706                  { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
707              }
```

Now, the main job.

```
708            \ttfamily
709            \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
```

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

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

```
725          \end { #1 }
726          \@@_write_aux:
727        }
```

We can now define the new environment.
We are still in the definition of the command `\NewPitonEnvironment`...

```
728      \NewDocumentEnvironment { #1 } { #2 }
729        {
730          #3
731          \@@_pre_env:
732          \group_begin:
733          \tl_map_function:nN
734            { \ \\ \{ \} \$ \& \# \^ \_ \% \~ \^^I }
735            \char_set_catcode_other:N
736          \use:c { _@@_collect_ #1 :w }
737        }
738        { #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).

```
739      \AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
740    }
```

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`.

```
741  \bool_if:NTF \c_@@_beamer_bool
742    {
743      \NewPitonEnvironment { Piton } { d < > }
744        {
745          \IfValueTF { #1 }
746            { \begin { uncoverenv } < #1 > }
747            { \begin { uncoverenv } }
748        }
749        { \end { uncoverenv } }
750    }
751    { \NewPitonEnvironment { Piton } { } { } { } }
```

6.2.8 The styles

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

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

The following command takes in its argument by curryfication.

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

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

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

```
813    Exception          .value_required:n = true ,
814    Comment.LaTeX      .tl_set:c = pitonStyle Comment.LaTeX ,
815    Comment.LaTeX      .value_required:n = true ,
816    Identifier         .tl_set:c = pitonStyle Identifier ,
817    Comment.LaTeX      .value_required:n = true ,
818    ParseAgain.noCR    .tl_set:c = pitonStyle ParseAgain.noCR ,
819    ParseAgain.noCR    .value_required:n = true ,
820    ParseAgain         .tl_set:c = pitonStyle ParseAgain ,
821    ParseAgain         .value_required:n = true ,
822    Prompt             .tl_set:c = pitonStyle Prompt ,
823    Prompt             .value_required:n = true ,
824    unknown            .code:n =
825       \msg_error:nn { piton } { Unknown~key~for~SetPitonStyle }
826    }


827  \msg_new:nnn { piton } { Unknown~key~for~SetPitonStyle }
828    {
829      The~style~'\l_keys_key_str'~is~unknown.\\
830      This~key~will~be~ignored.\\
831      The~available~styles~are~(in~alphabetic~order):~
832      Comment,~
833      Comment.LaTeX,~
834      Dict.Value,~
835      Exception,~
836      Identifier,~
837      InitialValues,~
838      Keyword,~
839      Keyword.Constant,~
840      Name.Builtin,~
841      Name.Class,~
842      Name.Constructor,~
843      Name.Decorator,~
844      Name.Function,~
845      Name.Module,~
846      Name.ModuleType,~
847      Name.Namespace,~
848      Name.UserFunction,~
849      Number,~
850      Operator,~
851      Operator.Word,~
852      Prompt,~
853      String,~
854      String.Doc,~
855      String.Long,~
856      String.Short,~and~
857      String.Interpol.
858    }
```

6.2.9 The initial style

The initial style is inspired by the style "manni" of Pygments.

```
859  \SetPitonStyle
860    {
861      Comment          = \color[HTML]{0099FF} \itshape ,
862      Exception        = \color[HTML]{CC0000} ,
863      Keyword          = \color[HTML]{006699} \bfseries ,
864      Keyword.Constant = \color[HTML]{006699} \bfseries ,
865      Name.Builtin     = \color[HTML]{336666} ,
866      Name.Decorator   = \color[HTML]{9999FF},
867      Name.Class       = \color[HTML]{00AA88} \bfseries ,
```

```
868    Name.Function      = \color[HTML]{CC00FF} ,
869    Name.Namespace     = \color[HTML]{00CCFF} ,
870    Name.Constructor   = \color[HTML]{006000} \bfseries ,
871    Name.Module        = \color[HTML]{0060A0} \bfseries ,
872    Name.ModuleType    = ,
873    Number             = \color[HTML]{FF6600} ,
874    Operator           = \color[HTML]{555555} ,
875    Operator.Word      = \bfseries ,
876    String             = \color[HTML]{CC3300} ,
877    String.Doc         = \color[HTML]{CC3300} \itshape ,
878    String.Interpol    = \color[HTML]{AA0000} ,
879    Comment.LaTeX      = \normalfont \color[rgb]{.468,.532,.6} ,
880    Name.Type          = \color[HTML]{336666} ,
881    InitialValues      = \@@_piton:n ,
882    Dict.Value         = \@@_piton:n ,
883    Interpol.Inside    = \color{black}\@@_piton:n ,
884    TypeParameter      = \color[HTML]{008800} \itshape ,
885    Identifier         = \@@_identifier:n ,
886    Name.UserFunction  = ,
887    Prompt             = ,
888    ParseAgain.noCR    = \@@_piton_no_cr:n ,
889    ParseAgain         = \@@_piton:n ,
890  }
```

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)].

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

6.2.10 Highlighting some identifiers

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


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


896  \keys_define:nn { Piton / identifiers }
897    {
898      names .clist_set:N = \l_@@_identifiers_names_tl ,
899      style .tl_set:N    = \l_@@_style_tl ,
900    }


901  \cs_new_protected:Npn \@@_set_identifiers:n #1
902    {
903      \clist_clear_new:N \l_@@_identifiers_names_tl
904      \tl_clear_new:N \l_@@_style_tl
905      \keys_set:nn { Piton / identifiers } { #1 }
906      \clist_map_inline:Nn \l_@@_identifiers_names_tl
907        {
908          \tl_set_eq:cN
909            { PitonIdentifier _ \l_@@_language_str _ ##1 }
910            \l_@@_style_tl
911        }
912    }
```

39

In particular, we have an highlighting of the indentifiers 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`).

```
913 \cs_new_protected:cpn { pitonStyle Name.Function.Internal } #1
914    {
```

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

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

Now, we specify that the name of the new Python function is a known identifier that will be formated 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}).

```
916       \cs_gset_protected:cpn { PitonIdentifier _ \l_@@_language_str _ #1 }
917          { \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.**

```
918       \seq_if_exist:cF { g_@@_functions _ \l_@@_language_str _ seq }
919          { \seq_new:c { g_@@_functions _ \l_@@_language_str _ seq } }
920       \seq_gput_right:cn { g_@@_functions _ \l_@@_language_str _ seq } { #1 }
921    }


922 \NewDocumentCommand \PitonClearUserFunctions { ! O { \l_@@_language_str }  }
923    {
924       \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
925          {
926             \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
927                { \cs_undefine:c { PitonIdentifier _ #1 _ ##1} }
928             \seq_gclear:c { g_@@_functions _ #1 _ seq }
929          }
930    }
```

6.2.11 Security

```
931 \AddToHook { env / piton / begin }
932    { \msg_fatal:nn { piton } { No~environment~piton } }
933
934 \msg_new:nnn { piton } { No~environment~piton }
935    {
936       There~is~no~environment~piton!\\
937       There~is~an~environment~{Piton}~and~a~command~
938       \token_to_str:N \piton\ but~there~is~no~environment~
939       {piton}.~This~error~is~fatal.
940    }
```

6.2.12 The error messages of the package

```
941 \msg_new:nnnn { piton } { Unknown~key~for~PitonOptions }
942    {
943       Unknown~key. \\
944       The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
945       It~will~be~ignored.\\
946       For~a~list~of~the~available~keys,~type~H~<return>.
947    }
948    {
949       The~available~keys~are~(in~alphabetic~order):~
950       all-line-numbers,~
951       auto-gobble,~
952       background-color,~
953       break-lines,~
954       break-lines-in-piton,~
955       break-lines-in-Piton,~
```

40

```
956    continuation-symbol,~
957    continuation-symbol-on-indentation,~
958    end-of-broken-line,~
959    env-gobble,~
960    gobble,~
961    identifiers,~
962    indent-broken-lines,~
963    language,~
964    left-margin,~
965    line-numbers,~
966    prompt-background-color,~
967    resume,~
968    show-spaces,~
969    show-spaces-in-strings,~
970    slim,~
971    splittable,~
972    tabs-auto-gobble~
973    and~tab-size.
974  }


975 \msg_new:nnn { piton } { label~with~lines~numbers }
976   {
977     You~can't~use~the~command~\token_to_str:N \label\
978     because~the~key~'line-numbers'~(or~'all-line-numbers')~
979     is~not~active.\\
980     If~you~go~on,~that~command~will~ignored.
981   }


982 \msg_new:nnn { piton } { cr~not~allowed }
983   {
984     You~can't~put~any~carriage~return~in~the~argument~
985     of~a~command~\c_backslash_str
986     \l_@@_beamer_command_str\ within~an~
987     environment~of~'piton'.~You~should~consider~using~the~
988     corresponding~environment.\\
989     That~error~is~fatal.
990   }


991 \msg_new:nnn { piton } { overlay~without~beamer }
992   {
993     You~can't~use~an~argument~<...>~for~your~command~
994     \token_to_str:N \PitonInputFile\ because~you~are~not~
995     in~Beamer.\\
996     If~you~go~on,~that~argument~will~be~ignored.
997   }


998 \msg_new:nnn { Piton } { Python~error }
999   { A~Python~error~has~been~detected. }
```

6.3 The Lua part of the implementation

```
1000 \ExplSyntaxOff
1001 \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`.

```
1002 \begin{luacode*}
1003 piton = piton or { }
```

```
1004  if piton.comment_latex == nil then piton.comment_latex = ">" end
1005  piton.comment_latex = "#" .. piton.comment_latex
```

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

```
1006  function piton.open_brace ()
1007     tex.sprint("{")
1008  end
1009  function piton.close_brace ()
1010     tex.sprint("}")
1011  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.

```
1012  local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
1013  local Cf, Cs , Cg , Cmt , Cb = lpeg.Cf, lpeg.Cs, lpeg.Cg , lpeg.Cmt , lpeg.Cb
1014  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").

```
1015  local function Q(pattern)
1016     return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
1017  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 beetween "escape-inside". That function won't be much used.

```
1018  local function L(pattern)
1019     return Ct ( C ( pattern ) )
1020  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.

```
1021  local function Lc(string)
1022     return Cc ( { luatexbase.catcodetables.expl , string } )
1023  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)

```
1024  local function K(style, pattern)
1025     return
1026        Lc ( "{\\PitonStyle{" .. style .. "}{" )
1027        * Q ( pattern )
1028        * Lc ( "}}" )
1029  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*`}}`.

```
1030  local function WithStyle(style,pattern)
1031    return
1032        Ct ( Cc "Open" * Cc ( "{\\PitonStyle{" .. style .. "}{" ) * Cc "}}" )
1033      * pattern
1034      * Ct ( Cc "Close" )
1035  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 catched must be sent to LaTeX with standard LaTeX catcodes, we put the capture (done by the function `C`) in a table (by using `Ct`, which is an alias for `lpeg.Ct`) without number of catcode table at the first component of the table.

```
1036  local Escape =
1037    P(piton_begin_escape)
1038    * L ( ( 1 - P(piton_end_escape) ) ^ 1 )
1039    * P(piton_end_escape)
```

The following line is mandatory.

```
1040  lpeg.locale(lpeg)
```

The basic syntactic LPEG

```
1041  local alpha, digit = lpeg.alpha, lpeg.digit
1042  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.

```
1043  local letter = alpha + P "_"
1044    + P "â" + P "à" + P "ç" + P "é" + P "è" + P "ê" + P "ë" + P "ï" + P "î"
1045    + P "ô" + P "û" + P "ü" + P "Â" + P "À" + P "Ç" + P "É" + P "È" + P "Ê"
1046    + P "Ë" + P "Ï" + P "Î" + P "Ô" + P "Û" + P "Ü"
1047
1048  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).

```
1049  local identifier = letter * alphanum ^ 0
```

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

```
1050  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).

---

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

```
1051  local Number =
1052    K ( 'Number' ,
1053        ( digit^1 * P "." * digit^0 + digit^0 * P "." * digit^1 + digit^1 )
1054        * ( S "eE" * S "+-" ^ -1 * digit^1 ) ^ -1
1055        + digit^1
1056      )
```

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.

```
1057  local Word
1058  if piton_begin_escape ~= ''
1059  then Word = Q ( ( ( 1 - space - P(piton_begin_escape) - P(piton_end_escape) )
1060                    - S "'\"\r[()]" - digit ) ^ 1 )
1061  else Word = Q ( ( ( 1 - space ) - S "'\"\r[()]" - digit ) ^ 1 )
1062  end


1063  local Space = ( Q " " ) ^ 1

1064
1065  local SkipSpace = ( Q " " ) ^ 0

1066
1067  local Punct = Q ( S ".,:;!" )

1068
1069  local Tab = P "\t" * Lc ( '\\l_@@_tab_tl' )


1070  local SpaceIndentation = Lc ( '\\@@_an_indentation_space:' ) * ( Q " " )


1071  local Delim = Q ( S "[()]" )
```

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

```
1072  local VisualSpace = space * Lc "\\l_@@_space_tl"
```

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.

```
1073  local Operator =
1074    K ( 'Operator' ,
1075        P "!=" + P "<>" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P ":="
1076        + P "//" + P "**" + S "-~+/*%=<>&.@|"
1077      )

1078
1079  local OperatorWord =
1080    K ( 'Operator.Word' ,P "in" + P "is" + P "and" + P "or" + P "not" )

1081
1082  local Keyword =
1083    K ( 'Keyword' ,
1084        P "as" + P "assert" + P "break" + P "case" + P "class" + P "continue"
1085        + P "def" + P "del" + P "elif" + P "else" + P "except" + P "exec"
1086        + P "finally" + P "for" + P "from" + P "global" + P "if" + P "import"
```

---

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

```
1087        + P "lambda" + P "non local" + P "pass" + P "return" + P "try"
1088        + P "while" + P "with" + P "yield" + P "yield from" )
1089    + K ( 'Keyword.Constant' ,P "True" + P "False" + P "None" )
1090
1091  local Builtin =
1092    K ( 'Name.Builtin' ,
1093        P "__import__" + P "abs" + P "all" + P "any" + P "bin" + P "bool"
1094      + P "bytearray" + P "bytes" + P "chr" + P "classmethod" + P "compile"
1095      + P "complex" + P "delattr" + P "dict" + P "dir" + P "divmod"
1096      + P "enumerate" + P "eval" + P "filter" + P "float" + P "format"
1097      + P "frozenset" + P "getattr" + P "globals" + P "hasattr" + P "hash"
1098      + P "hex" + P "id" + P "input" + P "int" + P "isinstance" + P "issubclass"
1099      + P "iter" + P "len" + P "list" + P "locals" + P "map" + P "max"
1100      + P "memoryview" + P "min" + P "next" + P "object" + P "oct" + P "open"
1101      + P "ord" + P "pow" + P "print" + P "property" + P "range" + P "repr"
1102      + P "reversed" + P "round" + P "set" + P "setattr" + P "slice" + P "sorted"
1103      + P "staticmethod" + P "str" + P "sum" + P "super" + P "tuple" + P "type"
1104      + P "vars" + P "zip" )
1105
1106
1107  local Exception =
1108    K ( 'Exception' ,
1109        P "ArithmeticError" + P "AssertionError" + P "AttributeError"
1110      + P "BaseException" + P "BufferError" + P "BytesWarning" + P "DeprecationWarning"
1111      + P "EOFError" + P "EnvironmentError" + P "Exception" + P "FloatingPointError"
1112      + P "FutureWarning" + P "GeneratorExit" + P "IOError" + P "ImportError"
1113      + P "ImportWarning" + P "IndentationError" + P "IndexError" + P "KeyError"
1114      + P "KeyboardInterrupt" + P "LookupError" + P "MemoryError" + P "NameError"
1115      + P "NotImplementedError" + P "OSError" + P "OverflowError"
1116      + P "PendingDeprecationWarning" + P "ReferenceError" + P "ResourceWarning"
1117      + P "RuntimeError" + P "RuntimeWarning" + P "StopIteration"
1118      + P "SyntaxError" + P "SyntaxWarning" + P "SystemError" + P "SystemExit"
1119      + P "TabError" + P "TypeError" + P "UnboundLocalError" + P "UnicodeDecodeError"
1120      + P "UnicodeEncodeError" + P "UnicodeError" + P "UnicodeTranslateError"
1121      + P "UnicodeWarning" + P "UserWarning" + P "ValueError" + P "VMSError"
1122      + P "Warning" + P "WindowsError" + P "ZeroDivisionError"
1123      + P "BlockingIOError" + P "ChildProcessError" + P "ConnectionError"
1124      + P "BrokenPipeError" + P "ConnectionAbortedError" + P "ConnectionRefusedError"
1125      + P "ConnectionResetError" + P "FileExistsError" + P "FileNotFoundError"
1126      + P "InterruptedError" + P "IsADirectoryError" + P "NotADirectoryError"
1127      + P "PermissionError" + P "ProcessLookupError" + P "TimeoutError"
1128      + P "StopAsyncIteration" + P "ModuleNotFoundError" + P "RecursionError" )
1129
1130
1131  local RaiseException = K ( 'Keyword' , P "raise" ) * SkipSpace * Exception * Q ( P "(" )
1132
```

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

```
1133  local Decorator = K ( 'Name.Decorator' , P "@" * letter^1  )
```

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

Example: `class myclass`:

```
1134  local DefClass =
1135    K ( 'Keyword' , P "class" ) * Space * K ( 'Name.Class' , identifier )
```

If the word `class` is not followed by a identifier, it will be catched as keyword by the LPEG `Keyword`
(useful if we want to type a list of keywords).

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

Example: `import numpy as np`

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

Example: `import math, numpy`

```
1136 local ImportAs =
1137   K ( 'Keyword' , P "import" )
1138     * Space
1139     * K ( 'Name.Namespace' ,
1140           identifier * ( P "." * identifier ) ^ 0 )
1141     * (
1142         ( Space * K ( 'Keyword' , P "as" ) * Space
1143           * K ( 'Name.Namespace' , identifier ) )
1144       +
1145         ( SkipSpace * Q ( P "," ) * SkipSpace
1146           * K ( 'Name.Namespace' , identifier ) ) ^ 0
1147     )
```

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

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

Example: `from math import pi`

```
1148 local FromImport =
1149   K ( 'Keyword' , P "from" )
1150     * Space * K ( 'Name.Namespace' , identifier )
1151     * 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 instruction[24] in that interpolation:
`f'Total price: {total+1:.2f} €'`

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

```
1152 local PercentInterpol =
1153   K ( 'String.Interpol' ,
1154       P "%"
1155       * ( P "(" * alphanum ^ 1 * P ")" ) ^ -1
1156       * ( S "-#0 +" ) ^ 0
1157       * ( digit ^ 1 + P "*" ) ^ -1
1158       * ( P "." * ( digit ^ 1 + P "*" ) ) ^ -1
1159       * ( S "HlL" ) ^ -1
1160       * S "sdfFeExXorgiGauc%"
1161   )
```

---

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

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]

```
1162 local SingleShortString =
1163    WithStyle ( 'String.Short' ,
```

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

```
1164          Q ( P "f'" + P "F'" )
1165          * (
1166             K ( 'String.Interpol' , P "{" )
1167              * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0  )
1168              * Q ( P ":" * (1 - S "}:'") ^ 0 ) ^ -1
1169              * K ( 'String.Interpol' , P "}" )
1170             +
1171             VisualSpace
1172             +
1173             Q ( ( P "\\'" + P "{{" + P "}}" + 1 - S " {}'" ) ^ 1 )
1174          ) ^ 0
1175          * Q ( P "'" )
1176       +
```

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

```
1177          Q ( P "'" + P "r'" + P "R'" )
1178          * ( Q ( ( P "\\'" + 1 - S " '\r%" ) ^ 1 )
1179             + VisualSpace
1180             + PercentInterpol
1181             + Q ( P "%" )
1182          ) ^ 0
1183          * Q ( P "'" ) )
1184
1185
1186 local DoubleShortString =
1187    WithStyle ( 'String.Short' ,
1188          Q ( P "f\"" + P "F\"" )
1189          * (
1190             K ( 'String.Interpol' , P "{" )
1191              * Q ( ( 1 - S "}\":" ) ^ 0 , 'Interpol.Inside' )
1192              * ( K ( 'String.Interpol' , P ":" ) * Q ( (1 - S "}:\"") ^ 0 ) ) ^ -1
1193              * K ( 'String.Interpol' , P "}" )
1194             +
1195             VisualSpace
1196             +
1197             Q ( ( P "\\\"" + P "{{" + P "}}" + 1 - S " {}\"" ) ^ 1 )
1198          ) ^ 0
1199          * Q ( P "\"" )
1200       +
1201          Q ( P "\"" + P "r\"" + P "R\"" )
1202          * ( Q ( ( P "\\\"" + 1 - S " \"\r%" ) ^ 1 )
1203             + VisualSpace
1204             + PercentInterpol
1205             + Q ( P "%" )
1206          ) ^ 0
1207          * Q ( P "\"" ) )
1208
1209 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

---

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

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).

```
1210 local BalancedBraces =
1211   P { "E" ,
1212       E =
1213           (
1214               P "{" * V "E" * P "}"
1215               +
1216               ShortString
1217               +
1218               ( 1 - S "{}" )
1219           ) ^ 0
1220     }
```

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

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

```
1266        OneBeamerEnvironment "uncoverenv"
1267      + OneBeamerEnvironment "onlyenv"
1268      + OneBeamerEnvironment "visibleenv"
1269      + OneBeamerEnvironment "invisibleenv"
1270      + OneBeamerEnvironment "alertenv"
1271      + OneBeamerEnvironment "actionenv"
1272      +
1273        L (
```

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

```
1274          ( P "\\alt" )
1275        * P "<" * (1 - P ">") ^ 0 * P ">"
1276        * P "{"
1277        )
1278        * K ( 'ParseAgain.noCR' , BalancedBraces )
1279        * L ( P "}{" )
1280        * K ( 'ParseAgain.noCR' , BalancedBraces )
1281        * L ( P "}" )
1282      +
1283        L (
```

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

```
1284          ( P "\\temporal" )
1285        * P "<" * (1 - P ">") ^ 0 * P ">"
1286        * P "{"
1287        )
1288        * K ( 'ParseAgain.noCR' , BalancedBraces )
1289        * L ( P "}{" )
1290        * K ( 'ParseAgain.noCR' , BalancedBraces )
1291        * L ( P "}{" )
1292        * K ( 'ParseAgain.noCR' , BalancedBraces )
1293        * L ( P "}" )
```

Now for the environemnts.

```
1294    BeamerBeginEnvironments =
1295        ( space ^ 0 *
1296          L
1297            (
1298              P "\\begin{" * BeamerNamesEnvironments * "}"
1299              * ( P "<" * ( 1 - P ">") ^ 0 * P ">" ) ^ -1
1300            )
1301          * P "\r"
1302        ) ^ 0
1303    BeamerEndEnvironments =
1304        ( space ^ 0 *
1305          L ( P "\\end{" * BeamerNamesEnvironments * P "}" )
1306          * P "\r"
1307        ) ^ 0
1308 end
```

EOL The following LPEG will detect the Python prompts when the user is typesetting an interactive session of Python (directly or through {pyconsole} of pyluatex). We have to detect that prompt twice. The first detection (called *hasty detection*) will be before the `\@@_begin_line:` because you want to trigger a special background color for that row (and, after the `\@@_begin_line:`, it's too late to change de background).

```
1309 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).

```
1310 local Prompt = K ( 'Prompt' , ( ( P ">>>" + P "..." ) * P " " ^ -1 ) ^ -1  )
```

The following LPEG `EOL` is for the end of lines.

```
1311  local EOL =
1312    P "\r"
1313    *
1314    (
1315      ( space^0 * -1 )
1316      +
```

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].

```
1317      Ct (
1318          Cc "EOL"
1319          *
1320          Ct (
1321              Lc "\\@@_end_line:"
1322              * BeamerEndEnvironments
1323              * BeamerBeginEnvironments
1324              * PromptHastyDetection
1325              * Lc "\\@@_newline: \\@@_begin_line:"
1326              * Prompt
1327          )
1328      )
1329    )
1330    *
1331    SpaceIndentation ^ 0
```

The long strings

```
1332  local SingleLongString =
1333    WithStyle ( 'String.Long' ,
1334      ( Q ( S "fF" * P "'''" )
1335        * (
1336            K ( 'String.Interpol' , P "{" )
1337            * K ( 'Interpol.Inside' , ( 1 - S "}:\r" - P "'''" ) ^ 0  )
1338            * Q ( P ":" * (1 - S "}:\r" - P "'''" ) ^ 0 ) ^ -1
1339            * K ( 'String.Interpol' , P "}"  )
1340          +
1341            Q ( ( 1 - P "'''" - S "{}'\r" ) ^ 1 )
1342          +
1343            EOL
1344        ) ^ 0
1345      +
1346        Q ( ( S "rR" ) ^ -1  * P "'''" )
1347        * (
1348            Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
1349          +
1350            PercentInterpol
1351          +
1352            P "%"
1353          +
1354            EOL
1355        ) ^ 0
1356      )
1357      * Q ( P "'''" ) )
1358
1359
1360  local DoubleLongString =
1361    WithStyle ( 'String.Long' ,
1362      (
```

---

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

```
1363          Q ( S "fF" * P "\"\"\"" )
1364        * (
1365          K ( 'String.Interpol', P "{"  )
1366            * K ( 'Interpol.Inside' , ( 1 - S "}:\r" - P "\"\"\"" ) ^ 0 )
1367            * Q ( P ":" * (1 - S "}:\r" - P "\"\"\"" ) ^ 0 ) ^ -1
1368            * K ( 'String.Interpol' , P "}"  )
1369          +
1370          Q ( ( 1 - P "\"\"\"" - S "{}\"\r" ) ^ 1 )
1371          +
1372          EOL
1373        ) ^ 0
1374      +
1375        Q ( ( S "rR" ) ^ -1  * P "\"\"\"" )
1376        * (
1377          Q ( ( 1 - P "\"\"\"" - S "%\r" ) ^ 1 )
1378          +
1379          PercentInterpol
1380          +
1381          P "%"
1382          +
1383          EOL
1384        ) ^ 0
1385      )
1386      * Q ( P "\"\"\"" )
1387  )

1388 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`).

```
1389 local StringDoc =
1390     K ( 'String.Doc' , P "\"\"\"" )
1391       * ( K ( 'String.Doc' , (1 - P "\"\"\"" - P "\r" ) ^ 0  ) * EOL
1392           * Tab ^ 0
1393         ) ^ 0
1394       * 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.

```
1395 local CommentMath =
1396   P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$"
1397
1398 local Comment =
1399   WithStyle ( 'Comment' ,
1400      Q ( P "#" )
1401      * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 )
1402   * ( EOL + -1 )
```

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

```
1403 local CommentLaTeX =
1404   P(piton.comment_latex)
1405   * Lc "{\\PitonStyle{Comment.LaTeX}{\\ignorespaces"
1406   * L ( ( 1 - P "\r" ) ^ 0 )
1407   * Lc "}}"
1408   * ( 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).

```
1409  local Expression =
1410    P { "E" ,
1411        E = ( 1 - S "{}()[]\r," ) ^ 0
1412            * (
1413                (    P "{" * V "F" * P "}"
1414                  + P "(" * V "F" * P ")"
1415                  + P "[" * V "F" * P "]" ) * ( 1 - S "{}()[]\r," ) ^ 0
1416              ) ^ 0 ,
1417        F = ( 1 - S "{}()[]\r\"'" ) ^ 0
1418            * ( (
1419                    P "'" * (P "\\'" + 1 - S"'\r" )^0 * P "'"
1420                  + P "\"" * (P "\\\"" + 1 - S"\"\r" )^0 * P "\""
1421                  + P "{"   * V "F" * P "}"
1422                  + P "(" * V "F" * P ")"
1423                  + P "[" * V "F" * P "]"
1424              ) * ( 1 - S "{}()[]\r\"'" ) ^ 0 ) ^ 0 ,
1425      }
```

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

<div align="center">

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

</div>

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`.

```
1426  local Param =
1427    SkipSpace * Identifier * SkipSpace
1428    * (
1429          K ( 'InitialValues' , P "=" * Expression )
1430        + Q ( P ":" ) * SkipSpace * K ( 'Name.Type' , letter^1  )
1431      ) ^ -1

1432  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`...

```
1433  local DefFunction =
1434    K ( 'Keyword' , P "def" )
1435    * Space
1436    * K ( 'Name.Function.Internal' , identifier )
1437    * SkipSpace
1438    * Q ( P "(" ) * Params * Q ( P ")" )
1439    * SkipSpace
1440    * ( 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.

```
1441    * K ( 'ParseAgain' , ( 1 - S ":\r" )^0  )
1442    * Q ( P ":" )
1443    * ( SkipSpace
1444        * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
1445        * Tab ^ 0
1446        * SkipSpace
1447        * StringDoc ^ 0 -- there may be additionnal docstrings
1448      ) ^ -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 catched as keyword by the LPEG `Keyword` (useful if, for example, the final user wants to speak of the keyword `def`).

The dictionaries of Python   We have LPEG 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!

```
1449 local ItemDict =
1450    ShortString * SkipSpace * Q ( P ":" ) * K ( 'Dict.Value' , Expression  )
1451
1452 local ItemOfSet = SkipSpace * ( ItemDict + ShortString ) * SkipSpace
1453
1454 local Set =
1455    Q ( P "{" )
1456    * ItemOfSet * ( Q ( P "," ) * ItemOfSet )  ^ 0
1457    * Q ( P "}" )
```

Miscellaneous

```
1458 local ExceptionInConsole = Exception *  Q ( ( 1 - P "\r" ) ^ 0 ) * EOL
```

The main LPEG   First, the main loop :

```
1459 MainLoopPython =
1460    (  ( space^1 * -1 )
1461        + EOL
1462        + Space
1463        + Tab
1464        + Escape
1465        + CommentLaTeX
1466        + Beamer
1467        + UserCommands
1468        + LongString
1469        + Comment
1470        + ExceptionInConsole
1471        + Set
1472        + Delim
```

`Operator` must be before `Punct`.

```
1473        + Operator
1474        + ShortString
1475        + Punct
1476        + FromImport
1477        + RaiseException
1478        + DefFunction
1479        + DefClass
1480        + Keyword * ( Space + Punct + Delim + EOL+ -1 )
1481        + Decorator
1482        + OperatorWord * ( Space + Punct + Delim + EOL+ -1 )
1483        + Builtin * ( Space + Punct + Delim + EOL+ -1 )
1484        + Identifier
1485        + Number
1486        + Word
1487    ) ^ 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].

```
1488  local python = P ( true )
1489
1490  python =
1491    Ct (
1492        ( ( space - P "\r" ) ^0 * P "\r" ) ^ -1
1493        * BeamerBeginEnvironments
1494        * PromptHastyDetection
1495        * Lc '\\@@_begin_line:'
1496        * Prompt
1497        * SpaceIndentation ^ 0
1498        * MainLoopPython
1499        * -1
1500        * Lc '\\@@_end_line:'
1501      )
1502  local languages = { }
1503  languages['python'] = python
```

6.3.3 The LPEG ocaml

```
1504  local Delim = Q ( P "[|" + P "|]" + S "[()]" )
1505  local Punct = Q ( S ",:;!" )
1506  local identifier =
1507    ( R "az" + P "_") * ( R "az" + R "AZ" + S "_'" + digit ) ^ 0
1508
1509  local cap_identifier = R "AZ" * ( R "az" + R "AZ" + S "_'" + digit ) ^ 0
1510
1511  local Identifier = K ( 'Identifier' , identifier )
1512  local Constructor = K ( 'Name.Constructor' , cap_identifier )
1513
1514  local ModuleType = K ( 'Name.ModuleType' , cap_identifier )
1515
1516  local ModuleUse =
1517    K ( 'Name.Module' , cap_identifier )
1518    * Q "."
1519    * ( Identifier + Constructor )
1520
1521  local Operator =
1522    K ( 'Operator' ,
1523        P "!=" + P "<>" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P ":="
1524        + P "||" + P "&&" + P "//" + P "**" + P ";;" + P "::" + P "->"
1525        + P "+." + P "-." + P "*." + P "/."
1526        + S "-~+/*%=<>&@|"
1527      )
1528
1529  local OperatorWord =
1530    K ( 'Operator.Word' ,
1531        P "and" + P "asr" + P "land" + P "lor" + P "lsl" + P "lxor"
1532        + P "mod" + P "or" )
1533
1534  local Keyword =
1535    K ( 'Keyword' ,
1536        P "assert" + P "as" + P "begin" + P "class" + P "constraint" + P "done"
1537    + P "downto" + P "do" + P "else" + P "end" + P "exception" + P "external"
1538    + P "false" + P "for" + P "function" + P "functor" + P "fun"  + P "if"
```

---

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

```
1539    + P "include" + P "inherit" + P "initializer" + P "in"  + P "lazy" + P "let"
1540    + P "match" + P "method" + P "module" + P "mutable" + P "new" + P "object"
1541    + P "of" + P "open" + P "private" + P "raise" + P "rec" + P "sig"
1542    + P "struct" + P "then" + P "to" + P "true" + P "try" + P "type"
1543    + P "value" + P "val" + P "virtual" + P "when" + P "while" + P "with" )
1544    + K ( 'Keyword.Constant' , P "true" + P "false" )
1545
1546
1547  local Builtin =
1548    K ( 'Name.Builtin' , P "not" + P "incr" + P "decr" + P "fst" + P "snd" )
```

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

```
1549  local Exception =
1550    K (   'Exception' ,
1551       P "Division_by_zero" + P "End_of_File" + P "Failure"
1552     + P "Invalid_argument" + P "Match_failure" + P "Not_found"
1553     + P "Out_of_memory" + P "Stack_overflow" + P "Sys_blocked_io"
1554     + P "Sys_error" + P "Undefined_recursive_module" )
```

The characters in OCaml

```
1555  local Char =
1556    K ( 'String.Short' , P "'" * ( ( 1 - P "'" ) ^ 0 + P "\\'" ) * P "'" )
```

Beamer

```
1557  local BalancedBraces =
1558    P { "E" ,
1559        E =
1560            (
1561              P "{" * V "E" * P "}"
1562              +
1563              P "\"" * ( 1 - S "\"" ) ^ 0 * P "\""  -- OCaml strings
1564              +
1565              ( 1 - S "{}" )
1566            ) ^ 0
1567      }
1568  if piton_beamer
1569  then
1570    Beamer =
1571        L  ( P "\\pause" * ( P "[" * (1 - P "]") ^ 0 * P "]" ) ^ -1 )
1572      +
1573      (   P "\\uncover"   * Lc ( '\\@@_beamer_command:n{uncover}' )
1574        + P "\\only"      * Lc ( '\\@@_beamer_command:n{only}' )
1575        + P "\\alert"     * Lc ( '\\@@_beamer_command:n{alert}' )
1576        + P "\\visible"   * Lc ( '\\@@_beamer_command:n{visible}' )
1577        + P "\\invisible" * Lc ( '\\@@_beamer_command:n{invisible}' )
1578        + P "\\action"    * Lc ( '\\@@_beamer_command:n{action}' )
1579      )
1580      *
1581      L ( ( P "<" * (1 - P ">") ^ 0 * P ">" ) ^ -1 * P "{" )
1582      * K ( 'ParseAgain.noCR' , BalancedBraces )
1583      * L ( P "}" )
1584      +
1585      L (
1586          ( P "\\alt" )
1587        * P "<" * (1 - P ">") ^ 0 * P ">"
1588        * P "{"
1589        )
1590      * K ( 'ParseAgain.noCR' , BalancedBraces )
1591      * L ( P "}{" )
1592      * K ( 'ParseAgain.noCR' , BalancedBraces )
```

```
1593        * L ( P "}" )
1594      +
1595        L (
1596            ( P "\\temporal" )
1597            * P "<" * (1 - P ">") ^ 0 * P ">"
1598            * P "{"
1599          )
1600        * K ( 'ParseAgain.noCR' , BalancedBraces )
1601        * L ( P "}{" )
1602        * K ( 'ParseAgain.noCR' , BalancedBraces )
1603        * L ( P "}{" )
1604        * K ( 'ParseAgain.noCR' , BalancedBraces )
1605        * L ( P "}" )
1606    BeamerBeginEnvironments =
1607        ( space ^ 0 *
1608          L
1609            (
1610              P "\\begin{" * BeamerNamesEnvironments * "}"
1611              * ( P "<" * ( 1 - P ">") ^ 0 * P ">" ) ^ -1
1612            )
1613        * P "\r"
1614        ) ^ 0
1615    BeamerEndEnvironments =
1616        ( space ^ 0 *
1617          L ( P "\\end{" * BeamerNamesEnvironments * P "}" )
1618        * P "\r"
1619        ) ^ 0
1620 end
```

EOL

```
1621 local EOL =
1622    P "\r"
1623    *
1624    (
1625      ( space^0 * -1 )
1626      +
1627      Ct (
1628          Cc "EOL"
1629          *
1630          Ct (
1631              Lc "\\@@_end_line:"
1632              * BeamerEndEnvironments
1633              * BeamerBeginEnvironments
1634              * PromptHastyDetection
1635              * Lc "\\@@_newline: \\@@_begin_line:"
1636              * Prompt
1637            )
1638        )
1639    )
1640    *
1641    SpaceIndentation ^ 0
1642 %
1643 % \paragraph{The strings}
1644 %
1645 % We need a pattern |string| without captures because it will be used within the
1646 % comments of OCaml.
1647 %    \begin{macrocode}
1648 local string =
1649        Q ( P "\"" )
1650      * (
1651          VisualSpace
```

56

```
1652          +
1653          Q ( ( 1 - S " \"\r" ) ^ 1 )
1654          +
1655          EOL
1656        ) ^ 0
1657      * Q ( P "\"" )
1658   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`.

```
1659   local ext = ( R "az" + P "_" ) ^ 0
1660   local open = "{" * Cg(ext, 'init') * "|"
1661   local close = "|" * C(ext) * "}"
1662   local closeeq =
1663     Cmt ( close * Cb('init'),
1664          function (s, i, a, b) return a==b end )
```

The LPEG `QuotedStringBis` will do the second analysis.

```
1665   local QuotedStringBis =
1666     WithStyle ( 'String.Long' ,
1667        (
1668          VisualSpace
1669          +
1670          Q ( ( 1 - S " \r" ) ^ 1 )
1671          +
1672          EOL
1673        ) ^ 0  )
1674
```

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.

```
1675   local QuotedString =
1676      C ( open * ( 1 - closeeq ) ^ 0  * close ) /
1677      ( 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).

```
1678   local Comment =
1679     WithStyle ( 'Comment' ,
1680        P {
1681          "A" ,
1682          A = Q "(*"
1683              * ( V "A"
1684                  + Q ( ( 1 - P "(*" - P "*)" - S "\r$\"" ) ^ 1 ) -- $
1685                  + string
1686                  + P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$" -- $
1687                  + EOL
1688                ) ^ 0
1689              * Q "*)"
1690        }   )
```

The DefFunction

```
1691 local DefFunction =
1692   K ( 'Keyword' , P "let rec" + P "let" + P "and" )
1693   * Space
1694   * K ( 'Name.Function.Internal' , identifier )
1695   * Space
1696   * # ( P "=" * space * P "function" + ( 1 - P "=" ) )
```

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

```
1697 local DefModule =
1698   K ( 'Keyword' , P "module" ) * Space
1699   *
1700     (
1701         K ( 'Keyword' , P "type" ) * Space
1702       * K ( 'Name.ModuleType' , cap_identifier )
1703     +
1704       K ( 'Name.Module' , cap_identifier ) * SkipSpace
1705       *
1706         (
1707           Q "(" * SkipSpace
1708           * K ( 'Name.Module' , cap_identifier ) * SkipSpace
1709           * Q ":" * SkipSpace
1710           * K ( 'Name.ModuleType' , cap_identifier ) * SkipSpace
1711           *
1712             (
1713               Q "," * SkipSpace
1714               * K ( 'Name.Module' , cap_identifier ) * SkipSpace
1715               * Q ":" * SkipSpace
1716               * K ( 'Name.ModuleType' , cap_identifier ) * SkipSpace
1717             ) ^ 0
1718           * Q ")"
1719         ) ^ -1
1720       *
1721         (
1722           Q "=" * SkipSpace
1723           * K ( 'Name.Module' , cap_identifier )  * SkipSpace
1724           * Q "("
1725           * K ( 'Name.Module' , cap_identifier ) * SkipSpace
1726           *
1727             (
1728               Q ","
1729               *
1730               K ( 'Name.Module' , cap_identifier ) * SkipSpace
1731             ) ^ 0
1732           * Q ")"
1733         ) ^ -1
1734     )
1735   +
1736   K ( 'Keyword' , P "include" + P "open" )
1737   * Space * K ( 'Name.Module' , cap_identifier )
```

The parameters of the types

```
1738 local TypeParameter = K ( 'TypeParameter' , P "'" * alpha * # ( 1 - P "'" ) )
```

The main LPEG   First, the main loop :

```
1739 MainLoopOCaml =
1740   ( ( space^1 * -1 )
1741      + EOL
1742      + Space
1743      + Tab
1744      + Escape
1745      + Beamer
1746      + TypeParameter
1747      + String + QuotedString + Char
1748      + Comment
1749      + Delim
1750      + Operator
1751      + Punct
1752      + FromImport
1753      + ImportAs
1754      + Exception
1755      + DefFunction
1756      + DefModule
1757      + Keyword * ( Space + Punct + Delim + EOL + -1 )
1758      + OperatorWord * ( Space + Punct + Delim + EOL + -1 )
1759      + Builtin * ( Space + Punct + Delim + EOL + -1 )
1760      + ModuleUse * ( Space + Punct + Delim + EOL + -1 )
1761      + Constructor * ( Space + Punct + Delim + EOL + -1 )
1762      + Identifier
1763      + Number
1764      + Word
1765   ) ^ 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].

```
1766 local ocaml = P ( true )
1767
1768 ocaml =
1769   Ct (
1770        ( ( space - P "\r" ) ^0 * P "\r" ) ^ -1
1771        * BeamerBeginEnvironments
1772        * Lc ( '\\@@_begin_line:' )
1773        * SpaceIndentation ^ 0
1774        * MainLoopOCaml
1775        * -1
1776        * Lc ( '\\@@_end_line:' )
1777      )
1778 languages['ocaml'] = ocaml
```

6.3.4   The function Parse

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

```
1779 function piton.Parse(language,code)
1780   local t = languages[language] : match ( code )
1781   local left_stack = {}
1782   local right_stack = {}
1783   for _ , one_item in ipairs(t)
```

---

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

```
1784   do
1785     if one_item[1] == "EOL"
1786     then
1787         for _ , s in ipairs(right_stack)
1788           do tex.sprint( s )
1789           end
1790         for _ , s in ipairs(one_item[2])
1791           do tex.tprint(s)
1792           end
1793         for _ , s in ipairs(left_stack)
1794           do tex.sprint( s )
1795           end
1796     else
1797         if one_item[1] == "Open"
1798         then
1799             tex.sprint( one_item[2] )
1800             table.insert(left_stack,one_item[2])
1801             table.insert(right_stack,one_item[3])
1802         else
1803             if one_item[1] == "Close"
1804             then
1805                 tex.sprint( right_stack[#right_stack] )
1806                 left_stack[#left_stack] = nil
1807                 right_stack[#right_stack] = nil
1808             else
1809                 tex.tprint(one_item)
1810             end
1811         end
1812     end
1813   end
1814 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.

```
1815 function piton.ParseFile(language,name,first_line,last_line)
1816   s = ''
1817   local i = 0
1818   for line in io.lines(name)
1819   do i = i + 1
1820     if i >= first_line
1821     then s = s .. '\r' .. line
1822     end
1823     if i >= last_line then break end
1824   end
1825   piton.Parse(language,s)
1826 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 `#`.

```
1827 function piton.ParseBis(language,code)
1828   local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
1829   return piton.Parse(language,s)
1830 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 syntaxic element. In that case, you have to remove the potential `\@@_breakable_space:` that have been inserted when the key `break-lines` is in force.

```
1831 function piton.ParseTer(language,code)
1832    local s = ( Cs ( ( P '\\@@_breakable_space:' / ' ' + 1 ) ^ 0 ) )
1833            : match ( code )
1834    return piton.Parse(language,s)
1835 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 if depends on the value of n.

```
1836 local function gobble(n,code)
1837    function concat(acc,new_value)
1838       return acc .. new_value
1839    end
1840    if n==0
1841    then return code
1842    else
1843        return Cf (
1844                Cc ( "" ) *
1845                ( 1 - P "\r" ) ^ (-n)  * C ( ( 1 - P "\r" ) ^ 0 )
1846                 * ( C ( P "\r" )
1847                 * ( 1 - P "\r" ) ^ (-n)
1848                 * C ( ( 1 - P "\r" ) ^ 0 )
1849                ) ^ 0 ,
1850                concat
1851            ) : match ( code )
1852    end
1853 end
```

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

```
1854 local function add(acc,new_value)
1855    return acc + new_value
1856 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.

```
1857 local AutoGobbleLPEG =
1858    ( space ^ 0 * P "\r" ) ^ -1
1859    * Cf (
1860            (
```

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

```
1861            ( P " " ) ^ 0 * P "\r"
1862            +
1863            Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1864            * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * P "\r"
1865        ) ^ 0
```

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

```
1866            *
1867            ( Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1868            * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,
1869            math.min
1870        )
```

The following LPEG is similar but works with the indentations.

```
1871 local TabsAutoGobbleLPEG =
1872     ( space ^ 0 * P "\r" ) ^ -1
1873     * Cf (
1874             (
1875               ( P "\t" ) ^ 0 * P "\r"
1876               +
1877               Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )
1878               * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * P "\r"
1879             ) ^ 0
1880             *
1881             ( Cf ( Cc(0) * ( P "\t" * Cc(1) ) ^ 0 , add )
1882             * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,
1883             math.min
1884         )
```

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.

```
1885 local EnvGobbleLPEG =
1886    ( ( 1 - P "\r" ) ^ 0 * P "\r" ) ^ 0
1887     * Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add ) * -1


1888 function piton.GobbleParse(language,n,code)
1889    if n==-1
1890    then n = AutoGobbleLPEG : match(code)
1891    else if n==-2
1892         then n = EnvGobbleLPEG : match(code)
1893         else if n==-3
1894              then n = TabsAutoGobbleLPEG : match(code)
1895              end
1896         end
1897    end
1898    piton.Parse(language,gobble(n,code))
1899 end
```

6.3.7 To count the number of lines

```
1900 function piton.CountLines(code)
1901    local count = 0
1902    for i in code : gmatch ( "\r" ) do count = count + 1 end
1903    tex.sprint(
1904        luatexbase.catcodetables.expl ,
1905        '\\int_set:Nn \\l_@@_nb_lines_int {' .. count .. '}' )
1906 end

1907 function piton.CountNonEmptyLines(code)
1908    local count = 0
1909    count =
1910    ( Cf (  Cc(0) *
1911            (
1912              ( P " " ) ^ 0 * P "\r"
1913              + ( 1 - P "\r" ) ^ 0 * P "\r" * Cc(1)
1914            ) ^ 0
1915            * (1 - P "\r" ) ^ 0 ,
1916          add
1917        ) * -1 ) : match (code)
1918    tex.sprint(
1919        luatexbase.catcodetables.expl ,
```

```
1920        '\\int_set:Nn \\l_@@_nb_non_empty_lines_int {' .. count .. '}' )
1921 end


1922 function piton.CountLinesFile(name)
1923    local count = 0
1924    for line in io.lines(name) do count = count + 1 end
1925    tex.sprint(
1926        luatexbase.catcodetables.expl ,
1927        '\\int_set:Nn \\l_@@_nb_lines_int {' .. count .. '}' )
1928 end


1929 function piton.CountNonEmptyLinesFile(name)
1930    local count = 0
1931    for line in io.lines(name)
1932    do if not ( ( ( P " " ) ^ 0 * -1 ) : match ( line ) )
1933        then count = count + 1
1934        end
1935    end
1936    tex.sprint(
1937        luatexbase.catcodetables.expl ,
1938        '\\int_set:Nn \\l_@@_nb_non_empty_lines_int {' .. count .. '}' )
1939 end

1940 \end{luacode*}
```

7 History

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

`https://tug.org/svn/texlive/trunk/Master/texmf-dist/tex/lualatex/piton/piton.sty`

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

Changes between versions 1.4 and 1.5

New key `numbers-sep`.

Changes between versions 1.3 and 1.4

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

Changes between versions 1.2 and 1.3

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

Changes between versions 1.1 and 1.2

New keys `break-lines-in-piton` and `break-lines-in-Piton`.
New key `show-spaces-in-string` and modification of the key `show-spaces`.
When the class beamer is used, the environements {uncoverenv}, {onlyenv}, {visibleenv} and {invisibleenv}

Changes between versions 1.0 and 1.1

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

Changes between versions 0.99 and 1.0

New key `tabs-auto-gobble`.

Changes between versions 0.95 and 0.99

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

Changes between versions 0.9 and 0.95

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

Changes between versions 0.8 and 0.9

New key `tab-size`.
Integer value for the key `splittable`.

Changes between versions 0.7 and 0.8

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

Changes between versions 0.6 and 0.7

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

Contents