

# The package `piton`\*

F. Pantigny  
fpantigny@wanadoo.fr

November 29, 2022

## Abstract

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

## 1 Presentation

The package `piton` uses the Lua library LPEG<sup>1</sup> for parsing Python listings and typeset them with syntactic highlighting. Since it uses Lua code, it works with `lualatex` only (and won't work with the other engines: `latex`, `pdflatex` and `xelatex`). It does not use external program and the compilation does not require `--shell-escape`. The compilation is very fast since all the parsing is done by the library LPEG, written in C.

Here is an exemple of code typeset by `piton`, with the environment `{Piton}`.

```
from math import pi

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

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

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

---

\*This document corresponds to the version 0.99 of `piton`, at the date of 2022/11/29.

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

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

## 2 Use of the package

### 2.1 Loading the package

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

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

### 2.2 The tools provided to the user

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

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

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

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

- The environment `{Piton}` should be used to typeset multi-lines code. Since it takes its argument in a verbatim mode, it can't be used within the argument of a LaTeX command. For sake of customization, it's possible to define new environments similar to the environment `{Piton}` with the command `\NewPitonEnvironment`: cf. 3.3 p. 5.
- The command `\PitonInputFile` is used to insert and typeset a whole external file.

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

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

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

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

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

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

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

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

Examples:

```
\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}
```

It's possible to use the command `\piton` in the arguments of a LaTeX command.<sup>3</sup>

- [Syntaxe `\piton|...|`](#)

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

Exemples :

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

## 3 Customization

### 3.1 The command `\PitonOptions`

The command `\PitonOptions` takes in as argument a comma-separated list of *key=value* pairs. The scope of the settings done by that command is the current TeX group.<sup>4</sup>

- The key `gobble` takes in as value a positive integer *n*: the first *n* characters are discarded (before the process of highlightning of the code) for each line of the environment `{Piton}`. These characters are not necessarily spaces.
- When the key `auto-gobble` is in force, the extension `piton` computes the minimal value *n* of the number of consecutive spaces beginning each (non empty) line of the environment `{Piton}` and applies `gobble` with that value of *n*.
- When the key `env-gobble` is in force, `piton` analyzes the last line of the environment `{Piton}`, that is to say the line which contains `\end{Piton}` and determines whether that line contains only spaces followed by the `\end{Piton}`. If we are in that situation, `piton` computes the number *n* of spaces on that line and applies `gobble` with that value of *n*. The name of that key comes from *environment gobble*: the effect of `gobble` is set by the position of the commands `\begin{Piton}` and `\end{Piton}` which delimit the current environment.
- With the key `line-numbers`, the *non empty* lines (and all the lines of the *docstrings*, even the empty ones) are numbered in the environments `{Piton}` and in the listings resulting from the use of `\PitonInputFile`.
- With the key `all-line-numbers`, *all* the lines are numbered, including the empty ones.
- With the key `resume` the counter of lines is not set to zero at the beginning of each environment `{Piton}` or use of `\PitonInputFile` as it is otherwise. That allows a numbering of the lines across several environments.
- The key `left-margin` corresponds to a margin on the left. That key may be useful in conjunction with the key `line-numbers` or the key `line-all-numbers` if one does not want the numbers in an overlapping position on the left.

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

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

---

<sup>3</sup>For example, it's possible to use the command `\piton` in a footnote. Example : `s = 'A string'`.

<sup>4</sup>We remind that an LaTeX environment is, in particular, a TeX group.

- When the key `show-spaces` is activated, the spaces in the short strings (that is to say those delimited by ' or ") are replaced by the character `□` (U+2423 : OPEN BOX). Of course, that character U+2423 must be present in the monospaced font which is used.<sup>5</sup>

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

```
\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}
```

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

The command `\PitonOptions` provides in fact several other keys which will be described further (see in particular the “Pages breaks and line breaks” p. 7).

### 3.2 The styles

The package `piton` provides the command `\SetPitonStyle` to customize the different styles used to format the syntactic elements of the Python listings. The customizations done by that command are limited to the current TeX group.<sup>6</sup>

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

<sup>5</sup>The package `piton` simply uses the current monospaced font. The best way to change that font is to use the command `\setmonofont` of `fontspec`.

<sup>6</sup>We remind that an LaTeX environment is, in particular, a TeX group.

These LaTeX instructions must be formatting instructions such as `\color{...}`, `\bfseries`, `\slshape`, etc. (the commands of this kind are sometimes called *semi-global* commands). It's also possible to put, *at the end of the list of instructions*, a LaTeX command taking exactly one argument.

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

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

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

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

The different styles are described in the table 1. The initial settings done by `piton` in `piton.sty` are inspired by the style `manni` de Pygments.<sup>7</sup>

### 3.3 Creation of new environments

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

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

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

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

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

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

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

If one wishes to format Python code in a box of `tcolorbox`, it's possible to define an environment `{Python}` with the following code:

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

## 4 Advanced features

### 4.1 Mechanisms to escape to LaTeX

The package `piton` provides several mechanisms for escaping to LaTeX:

- It's possible to compose comments entirely in LaTeX.
- It's possible to have the elements between `$` in the comments composed in LaTeX mathematical mode.
- It's also possible to insert LaTeX code almost everywhere in a Python listing.

---

<sup>7</sup>See: <https://pygments.org/styles/>. Remark that, by default, Pygments provides for its style `manni` a colored background whose color is the HTML color `#F0F3F3`.

### 4.1.1 The “LaTeX comments”

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

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

For example, with the following loading:

```
\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. 9](#)

### 4.1.2 The key “math-comments”

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

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

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

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

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

### 4.1.3 The mechanism “escape-inside”

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

```

\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        $\colorbox{yellow!50}{return n*fact(n-1)}$$
\end{Piton}

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

```

*Caution* : The escape to LaTeX allowed by the characters of `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).

## 4.2 Page breaks and line breaks

### 4.2.1 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 `background-color`), the pages breaks are allowed, as soon as the key `splittable` is in force.<sup>8</sup>

### 4.2.2 Line breaks

By default, the lines of the listings produced by `{Piton}` and `\PitonInputFile` are not breakable.

**New 0.99** There exist several keys (available in `\PitonOptions`) to allow and control such line breaks.

- The key `break-lines` activates the lines breaks. Only the spaces (even in the strings) are allowed break points.
- With the key `indent-broken-lines`, the indentation of a broken line is respected at carriage return.
- The key `end-of-broken-line` corresponds to the symbol placed at the end of a broken line. The initial value is: `\hspace*{0.5em}\textbackslash`.
- The key `continuation-symbol` corresponds to the symbol placed at each carriage return. The initial value is: `+\;`.
- The key `continuation-symbol-on-indentation` corresponds to the symbol placed at each carriage return, on the position of the indentation (only when the key `indent-broken-line` is in force). The initial value is: `hookrightarrow\;`.

---

<sup>8</sup>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`.

The following code has been composed in a `{minipage}` of width 12 cm with the following tuning:

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

```
def 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
```

### 4.3 Footnotes in the environments of piton

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

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

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

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

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

In this document, the package `piton` has been loaded with the option `footnotehyper`. For examples of notes, cf. 5.3, p. 10.

### 4.4 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`.

*Remark:* Unlike with the package `listings`, the key `gobble` and its variants (`auto-gobble` and `env-gobble`) are applied *before* the transformation of the characters of tabulation in spaces.

## 5 Examples

### 5.1 Line numbering

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

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

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



```

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

```

```

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

```

## 5.2 Formatting of the LaTeX comments

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

```

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

```

```

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

```

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

```

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

```

```

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

```

```

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

```

*appel récursif*

*autre appel récursif*

### 5.3 Notes in the listings

In order to be able to extract the notes (which are typeset with the command `\footnote`), the extension `piton` must be loaded with the key `footnote` or the key `footnotehyper` as explained in the section 4.3 p. 8. 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)9
    elif x > 1:
        return pi/2 - arctan(1/x)10
    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:

```

---

<sup>9</sup>First recursive call.

<sup>10</sup>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) )

```

---

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

<sup>b</sup>Second 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)11
    elif x > 1:
        return pi/2 - arctan(1/x)12
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )

```

## 5.4 An example of tuning of the styles

The graphical styles have been presented in the section 3.2, p. 4.

We present now an example of tuning of these styles adapted to the documents in black and white. We use the font *DejaVu Sans Mono*<sup>13</sup> specified by the command `\setmonofont` of `fontspec`.

```

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

\SetPitonStyle

```

---

<sup>11</sup>First recursive call.

<sup>12</sup>Second recursive call.

<sup>13</sup>See: <https://dejavu-fonts.github.io>

```

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

```

```
from math import pi
```

```

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

    n is the number of terms in the sum
    """
    if x < 0:
        return -arctan(-x) # appel récursif
    elif x > 1:
        return pi/2 - arctan(1/x)
        (we have used that arctan(x) + arctan(1/x) = 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.

```

\ExplSyntaxOn
\NewDocumentEnvironment { PitonExecute } { ! 0 { } }
{
  \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 <code>String.Short</code> and <code>String.Long</code>
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 : <code>in</code> , <code>is</code> , <code>and</code> , <code>or</code> and <code>not</code>
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 <code>def</code> )
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 <code>class</code> )
Exception	the names of the exceptions (eg: <code>SyntaxError</code> )
Comment	the comments beginning with #
Comment.LaTeX	the comments beginning by #>, which are composed in LaTeX by <code>piton</code> (and simply called “LaTeX comments” in this document)
Keyword.Constant	<code>True</code> , <code>False</code> and <code>None</code>
Keyword	the following keywords : <code>as</code> , <code>assert</code> , <code>break</code> , <code>case</code> , <code>continue</code> , <code>def</code> , <code>del</code> , <code>elif</code> , <code>else</code> , <code>except</code> , <code>exec</code> , <code>finally</code> , <code>for</code> , <code>from</code> , <code>global</code> , <code>if</code> , <code>import</code> , <code>lambda</code> , <code>non local</code> , <code>pass</code> , <code>raise</code> , <code>return</code> , <code>try</code> , <code>while</code> , <code>with</code> , <code>yield</code> , <code>yield from</code> .

## 6 Implementation

### 6.1 Introduction

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

In fact, all that job is done by a LPEG called `SyntaxPython`. That LPEG, when matched against the string of a Python listing, returns as capture a Lua table containing data to send to LaTeX. The only thing to do after will be to apply `tex.tprint` to each element of that table.<sup>14</sup>

Consider, for example, the following Python code:

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

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

```
{ "\\_piton_begin_line:" }a  
{ "{\PitonStyle{Keyword}{ " }"b  
{ luatexbase.catcodetables.CatcodeTableOtherc, "def" }  
{ "}}"  
{ luatexbase.catcodetables.CatcodeTableOther, " " }  
{ "{\PitonStyle{Name.Function}{ " }  
{ luatexbase.catcodetables.CatcodeTableOther, "parity" }  
{ "}}"  
{ luatexbase.catcodetables.CatcodeTableOther, "(" }  
{ luatexbase.catcodetables.CatcodeTableOther, "x" }  
{ luatexbase.catcodetables.CatcodeTableOther, ")" }  
{ luatexbase.catcodetables.CatcodeTableOther, ":" }  
{ "\\_piton_end_line: \\_piton_newline: \\_piton_begin_line:" }  
{ luatexbase.catcodetables.CatcodeTableOther, " " }  
{ "{\PitonStyle{Keyword}{ " }  
{ luatexbase.catcodetables.CatcodeTableOther, "return" }  
{ "}}"  
{ luatexbase.catcodetables.CatcodeTableOther, " " }  
{ luatexbase.catcodetables.CatcodeTableOther, "x" }  
{ "{\PitonStyle{Operator}{ " }  
{ luatexbase.catcodetables.CatcodeTableOther, "&" }  
{ "}}"  
{ "{\PitonStyle{Number}{ " }  
{ luatexbase.catcodetables.CatcodeTableOther, "2" }  
{ "}}"  
{ "\\_piton_end_line:" }
```

---

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

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

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

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`)

---

<sup>14</sup>Recall that `tex.tprint` takes in as argument a Lua table whose first component is a “catcode table” and the second element a string. The string will be sent to LaTeX with the regime of catcodes specified by the catcode table. If no catcode table is provided, the standard catcodes of LaTeX will be used.

```

\__piton_begin_line:{\PitonStyle{Keyword}{def}}
\__piton_end_line:\__piton_newline:
\__piton_begin_line:\__piton_end_line:\__piton_newline:
\__piton_end_line:\__piton_newline:
\__piton_end_line:\__piton_newline:
\__piton_end_line:\__piton_newline:

```

## 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   { The~package~'piton'~must~be~used~with~LuaLaTeX.\\ It~won't~be~loaded. }
10 \sys_if_engine luatex:F { \msg_critical:nn { piton } { LuaLaTeX-mandatory } }

11 \RequirePackage { luatexbase }

```

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

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

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

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

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

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

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

```

15 \keys_define:nn { piton / package }
16   {
17     footnote .bool_set:N = \c_@@_footnote_bool ,
18     footnotehyper .bool_set:N = \c_@@_footnotehyper_bool ,
19     escape-inside .tl_set:N = \c_@@_escape_inside_tl ,
20     escape-inside .initial:n = ,
21     comment-latex .code:n = { \lua_now:n { comment_latex = "#1" } } ,
22     comment-latex .value_required:n = true ,
23     math-comments .bool_set:N = \c_@@_math_comments_bool ,
24     math-comments .default:n = true ,
25     unknown .code:n = \msg_error:nn { piton } { unknown~key~for~package }
26   }
27 \msg_new:nnn { piton } { unknown-key-for-package }
28   {
29     Unknown~key.\\
30     You~have~used~the~key~'\l_keys_key_str'~but~the~only~keys~available~here~
31     are~'comment-latex',~'escape-inside',~'footnote',~'footnotehyper'~and~
32     'math-comments'.~Other~keys~are~available~in~\token_to_str:N \PitonOptions.\\
33     That~key~will~be~ignored.
34   }

```

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

```
35 \ProcessKeysOptions { piton / package }
```

```
36 \begingroup
```



```

37 \cs_new_protected:Npn \@@_set_escape_char:nn #1 #2
38 {
39   \lua_now:n { piton_begin_escape = "#1" }
40   \lua_now:n { piton_end_escape = "#2" }
41 }
42 \cs_generate_variant:Nn \@@_set_escape_char:nn { x x }
43 \@@_set_escape_char:xx
44 { \tl_head:V \c_@@_escape_inside_tl }
45 { \tl_tail:V \c_@@_escape_inside_tl }
46 \endgroup

47 \hook_gput_code:nnn { begindocument } { . }
48 {
49   \@ifpackageloaded { xcolor }
50   { }
51   { \msg_fatal:nn { piton } { xcolor~not~loaded } }
52 }

53 \msg_new:nnn { piton } { xcolor~not~loaded }
54 {
55   xcolor~not~loaded \\
56   The~package~'xcolor'~is~required~by~'piton'.\\
57   This~error~is~fatal.
58 }

59 \msg_new:nnn { piton } { footnote~with~footnotehyper~package }
60 {
61   Footnote~forbidden.\\
62   You~can't~use~the~option~'footnote'~because~the~package~
63   footnotehyper~has~already~been~loaded.~
64   If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
65   within~the~environments~of~piton~will~be~extracted~with~the~tools~
66   of~the~package~footnotehyper.\\
67   If~you~go~on,~the~package~footnote~won't~be~loaded.
68 }

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

78 \bool_if:NT \c_@@_footnote_bool
79 {

```

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

```

80   \@ifclassloaded { beamer }
81   { \bool_set_false:N \c_@@_footnote_bool }
82   {
83     \@ifpackageloaded { footnotehyper }
84     { \@Q_error:n { footnote~with~footnotehyper~package } }
85     { \usepackage { footnote } }
86   }
87 }

88 \bool_if:NT \c_@@_footnotehyper_bool
89 {

```

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

```

90   \@ifclassloaded { beamer }

```

```

91     { \bool_set_false:N \c_@@_footnote_bool }
92     {
93       \ifpackageloaded { footnote }
94       { \@_error:n { footnotehyper~with~footnote~package } }
95       { \usepackage { footnotehyper } }
96       \bool_set_true:N \c_@@_footnote_bool
97     }
98   }

```

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

## 6.2.2 Parameters and technical definitions

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

```

99 \int_new:N \l_@@_nb_lines_int

```

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

```

100 \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.

```

101 \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).

```

102 \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.

```

103 \int_new:N \l_@@_splittable_int

```

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

```

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

```

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

```

105 \str_new:N \l_@@_background_color_str

```

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

```

106 \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`.

```

107 \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`).

```

108 \int_new:N \g_@@_env_int

```

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

```

109 \bool_new:N \l_@@_break_lines_bool

```

```

110 \bool_new:N \l_@@_indent_broken_lines_bool

```

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

```

111 \tl_new:N \l_@@_continuation_symbol_tl

```

```

112 \tl_set:Nn \l_@@_continuation_symbol_tl { + }

```

```

113 % The following token list corresponds to the key
114 % |continuation-symbol-on-indentation|. The name has been shorten to |csoi|.
115 \tl_new:N \l_@@_csoi_tl
116 \tl_set:Nn \l_@@_csoi_tl { $ \hookrightarrow \; $ }

```

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

```

117 \tl_new:N \l_@@_end_of_broken_line_tl
118 \tl_set:Nn \l_@@_end_of_broken_line_tl { \hspace*{0.5em} \textbackslash }

```

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

```

119 \bool_new:N \l_@@_slim_bool

```

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

```

120 \dim_new:N \l_@@_left_margin_dim

```

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

```

121 \bool_new:N \l_@@_left_margin_auto_bool

```

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

```

122 \tl_new:N \l_@@_tab_tl

123 \cs_new_protected:Npn \@@_set_tab_tl:n #1
124 {
125   \tl_clear:N \l_@@_tab_tl
126   \prg_replicate:nn { #1 }
127   { \tl_put_right:Nn \l_@@_tab_tl { ~ } }
128 }
129 \@@_set_tab_tl:n { 4 }

```

The following integer corresponds to the key `gobble`.

```

130 \int_new:N \l_@@_gobble_int

131 \tl_new:N \l_@@_space_tl
132 \tl_set:Nn \l_@@_space_tl { ~ }

```

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

```

133 \int_new:N \g_@@_indentation_int

134 \cs_new_protected:Npn \@@_an_indentation_space:
135 { \int_gincr:N \g_@@_indentation_int }

```

### 6.2.3 Treatment of a line of code

In the contents provided by Lua, each line of the Python code will be surrounded by `\@@_begin_line:` and `\@@_end_line:`.

```

136 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
137 {
138   \int_gzero:N \g_@@_indentation_int

```

Be careful: there is curryfication in the following lines.

```

139   \bool_if:NTF \l_@@_slim_bool
140     { \hcoffin_set:Nn \l_tmpa_coffin }
141     {
142       \str_if_empty:NTF \l_@@_background_color_str
143         {
144           \vcoffin_set:Nnn \l_tmpa_coffin
145             { \dim_eval:n { \linewidth - \l_@@_left_margin_dim } }
146         }
147     {

```

```

148     \vcoffin_set:Nnn \l_tmpa_coffin
149     { \dim_eval:n { \linewidth - \l_@@_left_margin_dim - 0.5 em } }
150   }
151 }
152 {
153   \language = -1
154   \raggedright
155   \strut
156   \tl_set:Nn \l_tmpa_tl { #1 }

```

If the key `break-lines` is in force, we replace all the characters U+0032 (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.

```

157   \bool_if:NT \l_@@_break_lines_bool
158   {
159     \regex_replace_all:nnN
160     { \x20 }
161     { \c { @@_breakable_space: } }
162     \l_tmpa_tl
163   }
164   \l_tmpa_tl \strut \hfil
165 }
166 \hbox_set:Nn \l_tmpa_box
167 {
168   \skip_horizontal:N \l_@@_left_margin_dim
169   \bool_if:NT \l_@@_line_numbers_bool
170   {
171     \bool_if:NF \l_@@_all_line_numbers_bool
172     { \tl_if_empty:nF { #1 } }
173     \@@_print_number:
174   }
175   \str_if_empty:NF \l_@@_background_color_str
176   { \skip_horizontal:n { 0.5 em } }
177   \coffin_typeset:Nnnnn \l_tmpa_coffin T l \c_zero_dim \c_zero_dim
178 }

```

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

```

179   \dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_width_dim
180   { \dim_gset:Nn \g_@@_width_dim { \box_wd:N \l_tmpa_box } }
181   \box_set_dp:Nn \l_tmpa_box { \box_dp:N \l_tmpa_box + 1.25 pt }
182   \box_set_ht:Nn \l_tmpa_box { \box_ht:N \l_tmpa_box + 1.25 pt }
183   \tl_if_empty:NTF \l_@@_background_color_str
184   { \box_use_drop:N \l_tmpa_box }
185   {
186     \vbox_top:n
187     {
188       \hbox:n
189       {
190         \exp_args:NV \color \l_@@_background_color_str
191         \vrule height \box_ht:N \l_tmpa_box
192         depth \box_dp:N \l_tmpa_box
193         width \l_@@_width_on_aux_dim
194       }
195       \skip_vertical:n { - \box_ht_plus_dp:N \l_tmpa_box }
196       \box_set_wd:Nn \l_tmpa_box \l_@@_width_on_aux_dim
197       \box_use_drop:N \l_tmpa_box
198     }
199   }
200   \vspace { - 2.5 pt }
201 }

```

```

202 \cs_new_protected:Npn \@@_newline:
203 {
204   \int_gincr:N \g_@@_line_int

```

```

205 \int_compare:nNnT \g_@@_line_int > { \l_@@_splittable_int - 1 }
206 {
207   \int_compare:nNnT
208     { \l_@@_nb_lines_int - \g_@@_line_int } > \l_@@_splittable_int
209     {
210       \egroup
211       \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
212       \newline
213       \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
214       \vtop \bgroup
215     }
216   }
217 }

```

```

218 \cs_set_protected:Npn \@@_breakable_space:
219 {
220   \discretionary
221     { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
222     {
223       \hbox_overlap_left:n
224         {
225           {
226             \normalfont \footnotesize \color { gray }
227             \l_@@_continuation_symbol_tl
228           }
229           \skip_horizontal:n { 0.3 em }
230           \str_if_empty:NF \l_@@_background_color_str
231             { \skip_horizontal:n { 0.5 em } }
232         }
233       \bool_if:NT \l_@@_indent_broken_lines_bool
234       {
235         \hbox:n
236           {
237             \prg_replicate:nn { \g_@@_indentation_int } { ~ }
238             { \color { gray } \l_@@_csoi_tl }
239           }
240       }
241     }
242     { \hbox { ~ } }
243 }

```

## 6.2.4 PitonOptions

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

```

244 \bool_new:N \l_@@_line_numbers_bool
245 \bool_new:N \l_@@_all_line_numbers_bool

```

The following flag corresponds to the key `resume`.

```

246 \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.

```

247 \keys_define:nn { PitonOptions }
248 {
249   gobble           .int_set:N           = \l_@@_gobble_int ,
250   gobble           .value_required:n    = true ,
251   auto-gobble      .code:n              = \int_set:Nn \l_@@_gobble_int { -1 } ,
252   auto-gobble      .value_forbidden:n   = true ,
253   env-gobble       .code:n              = \int_set:Nn \l_@@_gobble_int { -2 } ,

```

```

254 env-gobble      .value_forbidden:n = true ,
255 line-numbers    .bool_set:N       = \l_@@_line_numbers_bool ,
256 line-numbers    .default:n        = true ,
257 all-line-numbers .code:n =
258   \bool_set_true:N \l_@@_line_numbers_bool
259   \bool_set_true:N \l_@@_all_line_numbers_bool ,
260 all-line-numbers .value_forbidden:n = true ,
261 resume          .bool_set:N       = \l_@@_resume_bool ,
262 resume          .value_forbidden:n = true ,
263 splittable      .int_set:N        = \l_@@_splittable_int ,
264 splittable      .default:n        = 1 ,
265 background-color .str_set:N       = \l_@@_background_color_str ,
266 background-color .value_required:n = true ,
267 slim            .bool_set:N       = \l_@@_slim_bool ,
268 slim            .default:n        = true ,
269 left-margin     .code:n =
270   \str_if_eq:nnTF { #1 } { auto }
271   {
272     \dim_zero:N \l_@@_left_margin_dim
273     \bool_set_true:N \l_@@_left_margin_auto_bool
274   }
275   { \dim_set:Nn \l_@@_left_margin_dim { #1 } } ,
276 left-margin     .value_required:n = true ,
277 tab-size        .code:n           = \@@_set_tab_tl:n { #1 } ,
278 tab-size        .value_required:n = true ,
279 show-spaces     .code:n           = \tl_set:Nn \l_@@_space_tl { \_ } , % U+2423
280 show-spaces     .value_forbidden:n = true ,
281 break-lines     .bool_set:N       = \l_@@_break_lines_bool ,
282 break-lines     .default:n        = true ,
283 indent-broken-lines .bool_set:N   = \l_@@_indent_broken_lines_bool ,
284 indent-broken-lines .default:n    = true ,
285 end-of-broken-line .tl_set:N      = \l_@@_end_of_broken_line_tl ,
286 end-of-broken-line .value_required:n = true ,
287 continuation-symbol .tl_set:N    = \l_@@_continuation_symbol_tl ,
288 continuation-symbol .value_required:n = true ,
289 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
290 continuation-symbol-on-indentation .value_required:n = true ,
291 unknown        .code:n =
292   \msg_error:nn { piton } { Unknown~key~for~PitonOptions }
293 }

294 \msg_new:nnnn { piton } { Unknown~key~for~PitonOptions }
295 {
296   Unknown~key. \
297   The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
298   It~will~be~ignored.\
299   For~a~list~of~the~available~keys,~type~H~<return>.
300 }
301 {
302   The~available~keys~are~(in~alphabetic~order):~
303   all-line-numbers,~
304   auto-gobble,~
305   break-lines,~
306   continuation-symbol,~
307   continuation-symbol-on-indentation,~
308   end-of-broken-line,~
309   env-gobble,~gobble,~
310   left-margin,~
311   indent-broken-lines,~
312   line-numbers,~
313   resume,~
314   show-spaces,~
315   slim,~

```

```

316 splittable-
317 and~tab-size.\\
318 }

```

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

```

319 \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`).

```

320 \int_new:N \g_@@_visual_line_int
321 \cs_new_protected:Npn \@@_print_number:
322 {
323   \int_gincr:N \g_@@_visual_line_int
324   \hbox_overlap_left:n
325   {
326     { \color { gray } \footnotesize \int_to_arabic:n \g_@@_visual_line_int }
327     \skip_horizontal:n { 0.4 em }
328   }
329 }

```

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

```

330 \cs_new_protected:Npn \@@_write_aux:
331 {
332   \tl_if_empty:NF \g_@@_aux_tl
333   {
334     \iow_now:Nn \@mainaux { \ExplSyntaxOn }
335     \iow_now:Nx \@mainaux
336     {
337       \tl_gset:cn { c_@@_ \int_use:N \g_@@_env_int _ tl }
338       { \exp_not:V \g_@@_aux_tl }
339     }
340     \iow_now:Nn \@mainaux { \ExplSyntaxOff }
341   }
342   \tl_gclear:N \g_@@_aux_tl
343 }
344 \cs_new_protected:Npn \@@_width_to_aux:
345 {
346   \bool_if:NT \l_@@_slim_bool
347   {
348     \str_if_empty:NF \l_@@_background_color_str
349     {
350       \tl_gput_right:Nx \g_@@_aux_tl
351       {
352         \dim_set:Nn \l_@@_width_on_aux_dim
353         { \dim_eval:n { \g_@@_width_dim + 0.5 em } }
354       }
355     }
356   }
357 }

```

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

```

358 \NewDocumentCommand { \piton } { }
359 { \peek_meaning:NTF \bgroup \@@_piton_standard \@@_piton_verbatim }

```

```

360 \NewDocumentCommand { \@@_piton_standard } { m }
361 {
362   \group_begin:
363   \ttfamily
364   \cs_set_eq:NN \ \ \c_backslash_str
365   \cs_set_eq:NN \% \c_percent_str
366   \cs_set_eq:NN \{ \c_left_brace_str
367   \cs_set_eq:NN \} \c_right_brace_str
368   \cs_set_eq:NN \$ \c_dollar_str
369   \cs_set_protected:Npn \@@_begin_line: { }
370   \cs_set_protected:Npn \@@_end_line: { }
371   \lua_now:n { piton.pitonParse(token.scan_string()) } { #1 }
372   \group_end:
373 }
374 \NewDocumentCommand { \@@_piton_verbatim } { v }
375 {
376   \group_begin:
377   \ttfamily
378   \cs_set_protected:Npn \@@_begin_line: { }
379   \cs_set_protected:Npn \@@_end_line: { }
380   \lua_now:n { piton.Parse(token.scan_string()) } { #1 }
381   \group_end:
382 }

```

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

```

383 \cs_new_protected:Npn \@@_piton:n #1
384 {
385   \group_begin:
386   \cs_set_protected:Npn \@@_begin_line: { }
387   \cs_set_protected:Npn \@@_end_line: { }
388   \lua_now:n { piton.Parse(token.scan_string()) } { #1 }
389   \group_end:
390 }

```

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

```

391 \cs_new:Npn \@@_pre_env:
392 {
393   \int_gincr:N \g_@@_env_int
394   \tl_gclear:N \g_@@_aux_tl
395   \cs_if_exist_use:c { c_@@_ _ \int_use:N \g_@@_env_int _ tl }
396   \dim_compare:nNnT \l_@@_width_on_aux_dim = \c_zero_dim
397     { \dim_set_eq:NN \l_@@_width_on_aux_dim \linewidth }
398   \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
399   \dim_gzero:N \g_@@_width_dim
400   \int_gzero:N \g_@@_line_int
401   \dim_zero:N \parindent
402   \dim_zero:N \lineskip
403 }
404 \keys_define:nn { PitonInputFile }
405 {
406   first-line .int_set:N = \l_@@_first_line_int ,
407   first-line .value_required:n = true ,
408   last-line .int_set:N = \l_@@_last_line_int ,
409   last-line .value_required:n = true ,
410 }
411 \NewDocumentCommand { \PitonInputFile } { 0 { } m }

```



```

412 {
413   \group_begin:
414     \int_zero_new:N \l_@@_first_line_int
415     \int_zero_new:N \l_@@_last_line_int
416     \int_set_eq:NN \l_@@_last_line_int \c_max_int
417     \keys_set:nn { PitonInputFile } { #1 }
418     \@@_pre_env:
419     \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.

```

420   \lua_now:n { piton.CountLinesFile(token.scan_argument()) } { #2 }
421   % If the final user has used both |left-margin=auto| and |line-numbers| or
422   % |all-line-numbers|, we have to compute the width of the maximal number of
423   % lines at the end of the composition of the listing to fix the correct value to
424   % |left-margin|.
425   % \begin{macrocode}
426   \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
427   {
428     \hbox_set:Nn \l_tmpa_box
429     {
430       \footnotesize
431       \bool_if:NTF \l_@@_all_line_numbers_bool
432       {
433         \int_to_arabic:n
434         { \g_@@_visual_line_int + \l_@@_nb_lines_int }
435       }
436       {
437         \lua_now:n
438         { piton.CountNonEmptyLinesFile(token.scan_argument()) }
439         { #2 }
440         \int_to_arabic:n
441         { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
442       }
443     }
444     \dim_set:Nn \l_@@_left_margin_dim { \box_wd:N \l_tmpa_box + 0.5em }
445   }

```

Now, the main job.

```

446   \ttfamily
447   \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
448   \vtop \bgroup
449   \lua_now:e
450   { piton.ParseFile(token.scan_argument(),
451     \int_use:N \l_@@_first_line_int ,
452     \int_use:N \l_@@_last_line_int )
453   }
454   { #2 }
455   \egroup
456   \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
457   \@@_width_to_aux:
458   \group_end:
459   \@@_write_aux:
460 }

```

```

461 \NewDocumentCommand { \NewPitonEnvironment } { m m m m }
462 {
463   \dim_zero:N \parindent

```

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.

```

464   \use:x
465   {
466     \cs_set_protected:Npn

```

```

467     \use:c { _@@_collect_ #1 :w }
468     ###1
469     \c_backslash_str end \c_left_brace_str #1 \c_right_brace_str
470   }
471   {
472     \group_end:
473     \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.

```

474     \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`.

```

475     \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
476     {
477       \bool_if:NTF \l_@@_all_line_numbers_bool
478       {
479         \hbox_set:Nn \l_tmpa_box
480         {
481           \footnotesize
482           \int_to_arabic:n
483           { \g_@@_visual_line_int + \l_@@_nb_lines_int }
484         }
485       }
486       {
487         \lua_now:n
488         { piton.CountNonEmptyLines(token.scan_argument()) }
489         { ##1 }
490         \hbox_set:Nn \l_tmpa_box
491         {
492           \footnotesize
493           \int_to_arabic:n
494           { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
495         }
496       }
497       \dim_set:Nn \l_@@_left_margin_dim
498       { \box_wd:N \l_tmpa_box + 0.5 em }
499     }

```

Now, the main job.

```

500     \ttfamily
501     \bool_if:NT \c_@@_footnote_bool { \begin { savenotes } }
502     \vtop \bgroup
503     \lua_now:e
504     {
505       piton.GobbleParse
506       ( \int_use:N \l_@@_gobble_int , token.scan_argument() )
507     }
508     { ##1 }
509     \vspace { 2.5 pt }
510     \egroup
511     \bool_if:NT \c_@@_footnote_bool { \end { savenotes } }
512     \@@_width_to_aux:

```

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

```

513     \end { #1 }
514     \@@_write_aux:
515   }

```

We can now define the new environment.

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

```

516     \NewDocumentEnvironment { #1 } { #2 }
517     {
518       #3
519       \@@_pre_env:

```

```

520     \group_begin:
521     \tl_map_function:nN
522     { \ \ \ \{ \} \$ \& \# \^ \_ \% \~ \^^I }
523     \char_set_catcode_other:N
524     \use:c { _@@_collect_ #1 :w }
525   }
526   { #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).

```

527     \AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
528   }

```

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

```

529 \NewPitonEnvironment { Piton } { } { } { }

```

## 6.2.8 The styles

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

```

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

```

The following command takes in its argument by curryfication.

```

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

```

```

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

```

```

534 \keys_define:nn { piton / Styles }
535 {
536   String.Interpol .tl_set:c = pitonStyle String.Interpol ,
537   String.Interpol .value_required:n = true ,
538   FormattingType .tl_set:c = pitonStyle FormattingType ,
539   FormattingType .value_required:n = true ,
540   Dict.Value      .tl_set:c = pitonStyle Dict.Value ,
541   Dict.Value      .value_required:n = true ,
542   Name.Decorator  .tl_set:c = pitonStyle Name.Decorator ,
543   Name.Decorator  .value_required:n = true ,
544   Name.Function   .tl_set:c = pitonStyle Name.Function ,
545   Name.Function   .value_required:n = true ,
546   Keyword         .tl_set:c = pitonStyle Keyword ,
547   Keyword         .value_required:n = true ,
548   Keyword.Constant .tl_set:c = pitonStyle Keyword.Constant ,
549   Keyword.constant .value_required:n = true ,
550   String.Doc      .tl_set:c = pitonStyle String.Doc ,
551   String.Doc      .value_required:n = true ,
552   Interpol.Inside .tl_set:c = pitonStyle Interpol.Inside ,
553   Interpol.Inside .value_required:n = true ,
554   String.Long     .tl_set:c = pitonStyle String.Long ,
555   String.Long     .value_required:n = true ,
556   String.Short    .tl_set:c = pitonStyle String.Short ,
557   String.Short    .value_required:n = true ,
558   String          .meta:n = { String.Long = #1 , String.Short = #1 } ,
559   Comment.Math    .tl_set:c = pitonStyle Comment.Math ,
560   Comment.Math    .default:n = \@@_math_scantokens:n ,
561   Comment.Math    .initial:n = ,
562   Comment         .tl_set:c = pitonStyle Comment ,
563   Comment         .value_required:n = true ,
564   InitialValues   .tl_set:c = pitonStyle InitialValues ,

```

```

565 InitialValues .value_required:n = true ,
566 Number .tl_set:c = pitonStyle Number ,
567 Number .value_required:n = true ,
568 Name.Namespace .tl_set:c = pitonStyle Name.Namespace ,
569 Name.Namespace .value_required:n = true ,
570 Name.Class .tl_set:c = pitonStyle Name.Class ,
571 Name.Class .value_required:n = true ,
572 Name.Builtin .tl_set:c = pitonStyle Name.Builtin ,
573 Name.Builtin .value_required:n = true ,
574 Name.Type .tl_set:c = pitonStyle Name.Type ,
575 Name.Type .value_required:n = true ,
576 Operator .tl_set:c = pitonStyle Operator ,
577 Operator .value_required:n = true ,
578 Operator.Word .tl_set:c = pitonStyle Operator.Word ,
579 Operator.Word .value_required:n = true ,
580 Post.Function .tl_set:c = pitonStyle Post.Function ,
581 Post.Function .value_required:n = true ,
582 Exception .tl_set:c = pitonStyle Exception ,
583 Exception .value_required:n = true ,
584 Comment.LaTeX .tl_set:c = pitonStyle Comment.LaTeX ,
585 Comment.LaTeX .value_required:n = true ,
586 unknown .code:n =
587 \msg_error:nn { piton } { Unknown-key-for-SetPitonStyle }
588 }

```

```

589 \msg_new:nnn { piton } { Unknown-key-for-SetPitonStyle }
590 {
591 The~style~'\l_keys_key_str'~is~unknown.\\
592 This~key~will~be~ignored.\\
593 The~available~styles~are~(in~alphabetic~order):~
594 Comment,~
595 Comment.LaTeX,~
596 Dict.Value,~
597 Exception,~
598 InitialValues,~
599 Keyword,~
600 Keyword.Constant,~
601 Name.Builtin,~
602 Name.Class,~
603 Name.Decorator,~
604 Name.Function,~
605 Name.Namespace,~
606 Number,~
607 Operator,~
608 Operator.Word,~
609 String,~
610 String.Doc,~
611 String.Long,~
612 String.Short,~and~
613 String.Interpol.
614 }

```

### 6.2.9 The initial style

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

```

615 \SetPitonStyle
616 {
617 Comment = \color[HTML]{0099FF} \itshape ,
618 Exception = \color[HTML]{CC0000} ,
619 Keyword = \color[HTML]{006699} \bfseries ,

```

```

620 Keyword.Constant = \color[HTML]{006699} \bfseries ,
621 Name.Builtin      = \color[HTML]{336666} ,
622 Name.Decorator    = \color[HTML]{9999FF},
623 Name.Class        = \color[HTML]{00AA88} \bfseries ,
624 Name.Function     = \color[HTML]{CC00FF} ,
625 Name.Namespace   = \color[HTML]{00CCFF} ,
626 Number           = \color[HTML]{FF6600} ,
627 Operator         = \color[HTML]{555555} ,
628 Operator.Word    = \bfseries ,
629 String           = \color[HTML]{CC3300} ,
630 String.Doc       = \color[HTML]{CC3300} \itshape ,
631 String.Interpol   = \color[HTML]{AA0000} ,
632 Comment.LaTeX    = \normalfont \color[rgb]{.468,.532,.6} ,
633 Name.Type        = \color[HTML]{336666} ,
634 InitialValues    = \@_piton:n ,
635 Dict.Value       = \@_piton:n ,
636 Interpol.Inside  = \color{black}\@_piton:n ,
637 Post.Function    = \@_piton:n ,
638 }

```

The last style `Post.Function` should be considered as an “internal style” (not available for the final user).

If the key `math-comments` has been used at load-time, we change the style `Comment.Math` which should be considered only at an “internal style”. However, maybe we will document in a future version the possibility to write change the style *locally* in a document)].

```

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

```

## 6.2.10 Security

```

641 \AddToHook { env / piton / begin }
642   { \msg_fatal:nn { piton } { No-environment~piton } }
643
644 \msg_new:nnn { piton } { No-environment~piton }
645   {
646     There-is-no-environment-piton!\!
647     There-is-an-environment~{Piton}~and-a-command~
648     \token_to_str:N \piton\ but~there-is-no-environment~
649     {piton}.~This-error-is-fatal.
650   }

```

## 6.3 The Lua part of the implementation

```

651 \ExplSyntaxOff
652 \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`.

```

653 \begin{luacode*}
654 piton = piton or { }
655 if piton.comment_latex == nil then piton.comment_latex = ">" end
656 piton.comment_latex = "#" .. piton.comment_latex

```

### 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.

```

657 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
658 local Cf, Cs = lpeg.Cf, lpeg.Cs

```

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”).

```
659 local function Q(pattern)
660   return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
661 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 “comment LateX” in the environments `{Piton}` and the elements between “`escape-inside`”. That function won’t be much used.

```
662 local function L(pattern)
663   return Ct ( C ( pattern ) )
664 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.

```
665 local function Lc(string)
666   return Cc ( { luatexbase.catcodetables.expl , string } )
667 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 pattern (that is to say a LPEG without capture) and the second element is a Lua string corresponding to the name of a `piton` style. If the seconde argument is not present, the function `K` behaves as the function `Q` does.

```
668 local function K(pattern, style)
669   if style
670   then
671     return
672     Lc ( "{\\PitonStyle{" .. style .. "}" )
673     * Q ( pattern )
674     * Lc ( "}" )
675   else
676     return Q ( pattern )
677   end
678 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}}`.

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_espape` and `piton.end_espape` are Lua strings corresponding to the key `escape-inside`<sup>15</sup>. Since the elements that will be caught must be sent to LaTeX with standard LaTeX catcodes, we put the capture (done by the function `C`) in a table (by using `Ct`, which is an alias for `lpeg.Ct`) without number of catcode table at the first component of the table.

```
679 local Escape =
680   P(piton_begin_espape)
681   * L ( ( 1 - P(piton_end_espape) ) ^ 1 )
682   * P(piton_end_espape)
```

---

<sup>15</sup>The `piton` key `escape-inside` is available at load-time only.

The following line is mandatory.

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

### 6.3.2 The LPEG SyntaxPython

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

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

```
685 local letter = alpha + P "_"
686   + P "â" + P "ã" + P "ç" + P "é" + P "è" + P "ê" + P "ë" + P "ï" + P "î"
687   + P "ô" + P "û" + P "ü" + P "À" + P "Ã" + P "Ç" + P "É" + P "È" + P "Ê"
688   + P "Ë" + P "Ï" + P "Î" + P "Û" + P "Ü" + P "Û"
689
690 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).

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

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

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

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

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

```
693 local Number =
694   K (
695     ( digit^1 * P "." * digit^0 + digit^0 * P "." * digit^1 + digit^1 )
696     * ( S "eE" * S "+-" ^ -1 * digit^1 ) ^ -1
697     + digit^1 ,
698     'Number'
699   )
```

We recall that `piton.begin_escape` and `piton.end_escape` are Lua strings corresponding to the key `escape-inside`<sup>16</sup>. Of course, if the final user has not used the key `escape-inside`, these strings are empty.

```
700 local Word
701 if piton_begin_escape ~= ''
702 then Word = K ( ( ( 1 - space - P(piton_begin_escape) - P(piton_end_escape) )
703                 - S "\"\r[()]" - digit ) ^ 1 )
704 else Word = K ( ( ( 1 - space ) - S "\"\r[()]" - digit ) ^ 1 )
705 end

706 local Space = K ( ( space - P "\r" ) ^ 1 )
707
708 local SkipSpace = K ( ( space - P "\r" ) ^ 0 )
709
710 local Punct = K ( S ".,;!" )

711 local Tab = P "\t" * Lc ( '\\\l_@@_tab_t1' )
```

---

<sup>16</sup>The `piton` key `escape-inside` is available at load-time only.

```

712 local SpaceIndentation =
713   Lc ( '\\@@_an_indentation_space:' ) * K " "

```

The following LPEG EOL is for the end of lines.

```

714 local EOL =
715   P "\r"
716   *
717   (
718     ( space0 * -1 )
719     +

```

We recall that each line in the Python code we have to parse will be sent back to LaTeX between a pair `\\@@_begin_line: - \\@@_end_line:`<sup>17</sup>.

```

720   Lc ( '\\@@_end_line: \\@@_newline: \\@@_begin_line:' )
721   )
722   *
723   SpaceIndentation ^ 0

```

```

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

```

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

```

725 local Operator =
726   K ( P "!=" + P "<>" + P "==" + P "<<" + P ">>" + P "<=" + P ">=" + P "!="
727     + P "/" + P "*" + S "--+/*%=<>&.@|"
728     ,
729     'Operator'
730   )
731
732 local OperatorWord =
733   K ( P "in" + P "is" + P "and" + P "or" + P "not" , 'Operator.Word' )
734
735 local Keyword =
736   K ( P "as" + P "assert" + P "break" + P "case" + P "class" + P "continue"
737     + P "def" + P "del" + P "elif" + P "else" + P "except" + P "exec"
738     + P "finally" + P "for" + P "from" + P "global" + P "if" + P "import"
739     + P "lambda" + P "non local" + P "pass" + P "return" + P "try"
740     + P "while" + P "with" + P "yield" + P "yield from" ,
741     'Keyword' )
742   + K ( P "True" + P "False" + P "None" , 'Keyword.Constant' )
743
744 local Builtin =
745   K ( P "__import__" + P "abs" + P "all" + P "any" + P "bin" + P "bool"
746     + P "bytearray" + P "bytes" + P "chr" + P "classmethod" + P "compile"
747     + P "complex" + P "delattr" + P "dict" + P "dir" + P "divmod"
748     + P "enumerate" + P "eval" + P "filter" + P "float" + P "format"
749     + P "frozenset" + P "getattr" + P "globals" + P "hasattr" + P "hash"
750     + P "hex" + P "id" + P "input" + P "int" + P "isinstance" + P "issubclass"
751     + P "iter" + P "len" + P "list" + P "locals" + P "map" + P "max"
752     + P "memoryview" + P "min" + P "next" + P "object" + P "oct" + P "open"
753     + P "ord" + P "pow" + P "print" + P "property" + P "range" + P "repr"
754     + P "reversed" + P "round" + P "set" + P "setattr" + P "slice" + P "sorted"
755     + P "staticmethod" + P "str" + P "sum" + P "super" + P "tuple" + P "type"
756     + P "vars" + P "zip" ,
757     'Name.Builtin' )
758
759 local Exception =
760   K ( "ArithmeticError" + P "AssertionError" + P "AttributeError"
761     + P "BaseException" + P "BufferError" + P "BytesWarning" + P "DeprecationWarning"

```

---

<sup>17</sup>Remember that the `\\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\\@@_begin_line:`



```

762 + P "EOFError" + P "EnvironmentError" + P "Exception" + P "FloatingPointError"
763 + P "FutureWarning" + P "GeneratorExit" + P "IOError" + P "ImportError"
764 + P "ImportWarning" + P "IndentationError" + P "IndexError" + P "KeyError"
765 + P "KeyboardInterrupt" + P "LookupError" + P "MemoryError" + P "NameError"
766 + P "NotImplementedError" + P "OSError" + P "OverflowError"
767 + P "PendingDeprecationWarning" + P "ReferenceError" + P "ResourceWarning"
768 + P "RuntimeError" + P "RuntimeWarning" + P "StopIteration"
769 + P "SyntaxError" + P "SyntaxWarning" + P "SystemError" + P "SystemExit"
770 + P "TabError" + P "TypeError" + P "UnboundLocalError" + P "UnicodeDecodeError"
771 + P "UnicodeEncodeError" + P "UnicodeError" + P "UnicodeTranslateError"
772 + P "UnicodeWarning" + P "UserWarning" + P "ValueError" + P "VMSError"
773 + P "Warning" + P "WindowsError" + P "ZeroDivisionError"
774 + P "BlockingIOError" + P "ChildProcessError" + P "ConnectionError"
775 + P "BrokenPipeError" + P "ConnectionAbortedError" + P "ConnectionRefusedError"
776 + P "ConnectionResetError" + P "FileExistsError" + P "FileNotFoundError"
777 + P "InterruptedError" + P "IsADirectoryError" + P "NotADirectoryError"
778 + P "PermissionError" + P "ProcessLookupError" + P "TimeoutError"
779 + P "StopAsyncIteration" + P "ModuleNotFoundError" + P "RecursionError" ,
780 'Exception' )
781
782 local RaiseException = K ( P "raise" , 'Keyword' ) * SkipSpace * Exception * K ( P "(" )
783
784 local ExceptionInConsole = Exception * K ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

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

```

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

```

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:`

```

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

```

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

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

Example: `import numpy as np`

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

Example: `import math, numpy`

```

788 local ImportAs =
789   K ( P "import" , 'Keyword' )
790   * Space
791   * K ( identifier * ( P "." * identifier ) ^ 0 ,
792       'Name.Namespace'
793   )
794   * (
795     ( Space * K ( P "as" , 'Keyword' ) * Space * K ( identifier , 'Name.Namespace' ) )
796     +
797     ( SkipSpace * K ( P "," ) * SkipSpace * K ( identifier , 'Name.Namespace' ) ) ^ 0
798   )

```

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

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

Example: `from math import pi`

```

799 local FromImport =
800   K ( P "from" , 'Keyword' )
801   * Space * K ( identifier , 'Name.Namespace' )
802   * Space * K ( P "import" , 'Keyword' )

```

**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"""

First, we define LPEG for the interpolations in the f-strings. Here is an example of a f-string with an interpolation and a format instruction<sup>18</sup> in that interpolation:

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

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

```

803 local SingleShortInterpol =
804   K ( P "{" , 'String.Interpol' )
805   * K ( ( 1 - S "}'" ) ^ 0 , 'Interpol.Inside' )
806   * K ( P ":" * ( 1 - S "}'" ) ^ 0 ) ^ -1
807   * K ( P "}" , 'String.Interpol' )
808
809 local DoubleShortInterpol =
810   K ( P "{" , 'String.Interpol' )
811   * K ( ( 1 - S "}'" ) ^ 0 , 'Interpol.Inside' )
812   * ( K ( P ":" , 'String.Interpol' ) * K ( ( 1 - S "}'" ) ^ 0 ) ) ^ -1
813   * K ( P "}" , 'String.Interpol' )
814
815 local SingleLongInterpol =
816   K ( P "{" , 'String.Interpol' )
817   * K ( ( 1 - S "}'" - P "'''" ) ^ 0 , 'Interpol.Inside' )
818   * K ( P ":" * ( 1 - S "}'" - P "'''" ) ^ 0 ) ^ -1
819   * K ( P "}" , 'String.Interpol' )
820
821 local DoubleLongInterpol =
822   K ( P "{" , 'String.Interpol' )
823   * K ( ( 1 - S "}'" - P "\"\"" ) ^ 0 , 'Interpol.Inside' )
824   * K ( P ":" * ( 1 - S "}'" - P "\"\"" ) ^ 0 ) ^ -1
825   * K ( P "}" , 'String.Interpol' )

```

The following LPEG catches a space (U+0032) and replace it by `\l_@@_space_t1`. It will be used in the short strings. Usually, `\l_@@_space_t1` will contain a space and therefore there won't be difference. However, when the key `show-spaces` is in force, `\l_@@_space_t1` will contain `␣` (U+2423) in order to visualize the spaces.

```
826 local VisualSpace = P " " * Lc "\\l_@@_space_t1"
```

Now, we define LPEG for the parts of the strings which are *not* in the interpolations.

```

827 local SingleShortPureString =
828   ( K ( ( P "\"" + P "{" + P "}" ) + 1 - S "}'" ) ^ 1 ) + VisualSpace ) ^ 1
829

```

<sup>18</sup>There is no special python style for the formatting instruction (after the comma): the style which will be applied will be the style of the encompassing string, that is to say `String.Short` or `String.Long`.

```

830 local DoubleShortPureString =
831   ( K ( ( P "\\\"" + P "{" + P "}" + 1 - S " {}\" ) ^ 1 ) + VisualSpace ) ^ 1
832
833 local SingleLongPureString =
834   K ( ( 1 - P "'" - S "{}'\r" ) ^ 1 )
835
836 local DoubleLongPureString =
837   K ( ( 1 - P "\\\" - S "{}\"r" ) ^ 1 )

```

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

```

838 local PercentInterpol =
839   K ( P "%"
840     * ( P "(" * alphanum ^ 1 * P ")" ) ^ -1
841     * ( S "-#0 +" ) ^ 0
842     * ( digit ^ 1 + P "*" ) ^ -1
843     * ( P "." * ( digit ^ 1 + P "*" ) ) ^ -1
844     * ( S "HLL" ) ^ -1
845     * S "sdfFeExXorgiGauc%" ,
846     'String.Interpol'
847   )

```

We can now define the LPEG for the four kinds of strings. It's not possible to use our function K because of the interpolations which must be formatted with another piton style that the rest of the string.<sup>19</sup>

```

848 local SingleShortString =
849   Lc ( "{\\PitonStyle{String.Short}{" )
850     * (

```

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

```

851     K ( P "f'" + P "F'" )
852     * ( SingleShortInterpol + SingleShortPureString ) ^ 0
853     * K ( P "'" )
854   +

```

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

```

855     K ( P '"' + P "r'" + P "R'" )
856     * ( K ( ( P "\\\"" + 1 - S " '\r%" ) ^ 1 )
857         + VisualSpace
858         + PercentInterpol
859         + K ( P "%" )
860     ) ^ 0
861     * K ( P "'" )
862   )
863   * Lc ( "}" )
864
865 local DoubleShortString =
866   Lc ( "{\\PitonStyle{String.Short}{" )
867     * (
868     K ( P "f\"" + P "F\"" )
869     * ( DoubleShortInterpol + DoubleShortPureString ) ^ 0
870     * K ( P "\"" )
871   +
872     K ( P "\" + P "r\"" + P "R\"" )
873     * ( K ( ( P "\\\"" + 1 - S " \"\r%" ) ^ 1 )
874         + VisualSpace
875         + PercentInterpol
876         + K ( P "%" )
877     ) ^ 0
878     * K ( P "\" )

```

<sup>19</sup>The interpolations are formatted with the piton style `Interpol`. Inside. The initial value of that style is `\\@@_piton:n` wich means that the interpolations are parsed once again by piton.

```

879     )
880     * Lc ( "}" )
881
882
883 local ShortString = SingleShortString + DoubleShortString

```

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

```

884 local SingleLongString =
885   Lc "{\\PitonStyle{String.Long}{"
886   * (
887     K ( S "fF" * P "'''" )
888     * ( SingleLongInterpol + SingleLongPureString ) ^ 0
889     * Lc "}"
890     * (
891       EOL
892       +
893       Lc "{\\PitonStyle{String.Long}{"
894       * ( SingleLongInterpol + SingleLongPureString ) ^ 0
895       * Lc "}"
896       * EOL
897     ) ^ 0
898     * Lc "{\\PitonStyle{String.Long}{"
899     * ( SingleLongInterpol + SingleLongPureString ) ^ 0
900   +
901   K ( ( S "rR" ) ^ -1 * P "'''"
902     * ( 1 - P "'''" - P "\r" ) ^ 0 )
903   * Lc "}"
904   * (
905     Lc "{\\PitonStyle{String.Long}{"
906     * K ( ( 1 - P "'''" - P "\r" ) ^ 0 )
907     * Lc "}"
908     * EOL
909   ) ^ 0
910   * Lc "{\\PitonStyle{String.Long}{"
911   * K ( ( 1 - P "'''" - P "\r" ) ^ 0 )
912   )
913   * K ( P "'''" )
914   * Lc "}"
915
916
917 local DoubleLongString =
918   Lc "{\\PitonStyle{String.Long}{"
919   * (
920     K ( S "fF" * P "\\\"\\\" )
921     * ( DoubleLongInterpol + DoubleLongPureString ) ^ 0
922     * Lc "}"
923     * (
924       EOL
925       +
926       Lc "{\\PitonStyle{String.Long}{"
927       * ( DoubleLongInterpol + DoubleLongPureString ) ^ 0
928       * Lc "}"
929       * EOL
930     ) ^ 0
931     * Lc "{\\PitonStyle{String.Long}{"
932     * ( DoubleLongInterpol + DoubleLongPureString ) ^ 0
933   +
934   K ( ( S "rR" ) ^ -1 * P "\\\"\\\"
935     * ( 1 - P "\\\"\\\" - P "\r" ) ^ 0 )
936   * Lc "}"
937   * (
938     Lc "{\\PitonStyle{String.Long}{"

```

```

939         * K ( ( 1 - P "\"\\\"" - P "\r" ) ^ 0 )
940         * Lc "}"
941         * EOL
942     ) ^ 0
943     * Lc "{\PitonStyle{String.Long}{"
944     * K ( ( 1 - P "\"\\\"" - P "\r" ) ^ 0 )
945 )
946 * K ( P "\"\\\"" )
947 * Lc "}"
948 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`).

```

949 local StringDoc =
950     K ( P "\"\\\"" , 'String.Doc' )
951     * ( K ( ( 1 - P "\"\\\"" - P "\r" ) ^ 0 , 'String.Doc' ) * EOL * Tab ^ 0 ) ^ 0
952     * K ( ( 1 - P "\"\\\"" - P "\r" ) ^ 0 * P "\"\\\"" , 'String.Doc' )

```

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

```

953 local CommentMath =
954     P "$" * K ( ( 1 - S "$\r" ) ^ 1 , 'Comment.Math' ) * P "$"
955
956 local Comment =
957     Lc ( "{\PitonStyle{Comment}{" )
958     * K ( P "#" )
959     * ( CommentMath + K ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0
960     * Lc ( "}" )
961     * ( EOL + -1 )

```

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

```

962 local CommentLaTeX =
963     P(piton.comment_latex)
964     * Lc "{\PitonStyle{Comment.LaTeX}{\ignorespaces"
965     * L ( ( 1 - P "\r" ) ^ 0 )
966     * Lc "}"
967     * ( EOL + -1 )

```

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

```

968 local Expression =
969     P { "E" ,
970         E = ( 1 - S "{}() []\r," ) ^ 0
971         * (
972             ( P "{" * V "F" * P "}"
973               + P "(" * V "F" * P ")"
974               + P "[" * V "F" * P "]" ) * ( 1 - S "{}() []\r," ) ^ 0
975             ) ^ 0 ,
976         F = ( 1 - S "{}() []\r\''" ) ^ 0
977         * ( (
978             P "'" * (P "\\'" + 1 - S''\r" )^0 * P "'"
979             + P "\"" * (P "\\\"" + 1 - S"\r" )^0 * P "\""
980             + P "{" * V "F" * P "}"

```

```

981         + P "(" * V "F" * P ")"
982         + P "[" * V "F" * P "]"
983     ) * ( 1 - S "{ } ( [ ] \r \'"'"' ) ^ 0 ) ^ 0 ,
984 }

```

We will now define a LPEG `Params` that will catch the list of parameters (that is to say the *argspec*) in the definition of a Python function. For example, in the line of code

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

the LPEG `Params` will be used to catch the chunk `a,b,x=10,n:int`.

Of course, a `Params` is simply a comma-separated list of `Param`, and that's why we define first the LPEG `Param`.

```

985 local Param =
986   SkipSpace * Identifier * SkipSpace
987   * (
988     K ( P "=" * Expression , 'InitialValues' )
989     + K ( P ":" ) * SkipSpace * K ( letter^1 , 'Name.Type' )
990   ) ^ -1
991 local Params = ( Param * ( K "," * Param ) ^ 0 ) ^ -1

```

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

```

992 local DefFunction =
993   K ( P "def" , 'Keyword' )
994   * Space
995   * K ( identifier , 'Name.Function' )
996   * SkipSpace
997   * K ( P "(" ) * Params * K ( P ")" )
998   * SkipSpace
999   * ( K ( P "->" ) * SkipSpace * K ( identifier , 'Name.Type' ) ) ^ -1

```

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

```

1000 * K ( ( 1 - S ":\r" )^0 , 'Post.Function' )
1001 * K ( P ":" )
1002 * ( SkipSpace
1003   * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
1004   * Tab ^ 0
1005   * SkipSpace
1006   * StringDoc ^ 0 -- there may be additionnal docstrings
1007 ) ^ -1

```

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

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

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

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

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

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

```
1008 local ItemDict =
1009   ShortString * SkipSpace * K ( P ":" ) * K ( Expression , 'Dict.Value' )
1010
1011 local ItemOfSet = SkipSpace * ( ItemDict + ShortString ) * SkipSpace
1012
1013 local Set =
1014   K ( P "{" )
1015   * ItemOfSet * ( K ( P "," ) * ItemOfSet ) ^ 0
1016   * K ( P "}" )
```

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

```
1017 local SyntaxPythonAux =
```

We recall that each line in the Python code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>20</sup>.

```
1018   Lc ( '\\@@_begin_line:' ) *
1019   ( ( space - P "\r" ) ^ 0 * P "\r" ) ^ -1 *
1020   SpaceIndentation ^ 0 *
1021   ( ( space ^ 1 * -1 )
1022     + EOL
1023     + Tab
1024     + Space
1025     + Escape
1026     + CommentLaTeX
1027     + LongString
1028     + Comment
1029     + ExceptionInConsole
1030     + Set
1031     + Delim
```

**Operator** must be before **Punct**.

```
1032     + Operator
1033     + ShortString
1034     + Punct
1035     + FromImport
1036     + ImportAs
1037     + RaiseException
1038     + DefFunction
1039     + DefClass
1040     + Keyword * ( Space + Punct + Delim + EOL + -1)
1041     + Decorator
1042     + OperatorWord * ( Space + Punct + Delim + EOL + -1)
1043     + Builtin * ( Space + Punct + Delim + EOL + -1)
1044     + Identifier
1045     + Number
1046     + Word
1047   ) ^ 0 * -1 * Lc ( '\\@@_end_line:' )
```

---

<sup>20</sup>Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

We have written an auxiliary LPEG `SyntaxPythonAux` for legibility only.

```
1048 local SyntaxPython = Ct ( SyntaxPythonAux )
```

### 6.3.3 The function `Parse`

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

```
1049 function piton.Parse(code)
1050   local t = SyntaxPython : match ( code ) -- match is a method of the LPEG
1051   for _ , s in ipairs(t) do tex.tprint(s) end
1052 end
```

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

```
1053 function piton.pitonParse(code)
1054   local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
1055   return piton.Parse(s)
1056 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.

```
1057 function piton.ParseFile(name,first_line,last_line)
1058   s = ''
1059   local i = 0
1060   for line in io.lines(name)
1061     do i = i + 1
1062       if i >= first_line
1063         then s = s .. '\r' .. line
1064         end
1065       if i >= last_line then break end
1066     end
1067   piton.Parse(s)
1068 end
```

### 6.3.4 The preprocessors of the function `Parse`

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

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

```
1069 local function gobble(n,code)
1070   function concat(acc,new_value)
1071     return acc .. new_value
1072   end
1073   if n==0
1074     then return code
1075     else
1076       return Cf (
1077         Cc ( "" ) *
1078         ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
1079         * ( C ( P "\r" )
1080         * ( 1 - P "\r" ) ^ (-n)
1081         * C ( ( 1 - P "\r" ) ^ 0 )
1082         ) ^ 0 ,
```



```

1083             concat
1084             ) : match ( code )
1085     end
1086 end

```

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

```

1087 local function add(acc,new_value)
1088     return acc + new_value
1089 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.

```

1090 local AutoGobbleLPEG =
1091     ( space ^ 0 * P "\r" ) ^ -1
1092     * Cf (
1093         (

```

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

```

1094             ( P " " ) ^ 0 * P "\r"
1095             +
1096             Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1097             * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * P "\r"
1098             ) ^ 0

```

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

```

1099         *
1100         ( Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add )
1101         * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1 ,
1102         math.min
1103     )

```

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

```

1104 local EnvGobbleLPEG =
1105     ( ( 1 - P "\r" ) ^ 0 * P "\r" ) ^ 0
1106     * Cf ( Cc(0) * ( P " " * Cc(1) ) ^ 0 , add ) * -1

```

```

1107 function piton.GobbleParse(n,code)
1108     if n==-1
1109     then n = AutoGobbleLPEG : match(code)
1110     else if n==-2
1111     then n = EnvGobbleLPEG : match(code)
1112     end
1113     end
1114     piton.Parse(gobble(n,code))
1115 end

```

### 6.3.5 To count the number of lines

```

1116 function piton.CountLines(code)
1117     local count = 0
1118     for i in code : gmatch ( "\r" ) do count = count + 1 end
1119     tex.sprint(
1120         luatexbase.catcodetables.expl ,
1121         '\\int_set:Nn \\l_@@_nb_lines_int {' .. count .. '} ' )
1122 end

```

```

1123 function piton.CountNonEmptyLines(code)
1124   local count = 0
1125   count =
1126   ( Cf ( Cc(0) *
1127     (
1128       ( P " " ) ^ 0 * P "\r"
1129       + ( 1 - P "\r" ) ^ 0 * P "\r" * Cc(1)
1130     ) ^ 0
1131     * ( 1 - P "\r" ) ^ 0 ,
1132     add
1133     ) * -1 ) : match (code)
1134   tex.sprint(
1135     luatexbase.catcodetables.expl ,
1136     '\\int_set:Nn \\l_@@_nb_non_empty_lines_int {' .. count .. '}' )
1137 end

1138 function piton.CountLinesFile(name)
1139   local count = 0
1140   for line in io.lines(name) do count = count + 1 end
1141   tex.sprint(
1142     luatexbase.catcodetables.expl ,
1143     '\\int_set:Nn \\l_@@_nb_lines_int {' .. count .. '}' )
1144 end

1145 function piton.CountNonEmptyLinesFile(name)
1146   local count = 0
1147   for line in io.lines(name)
1148   do if not ( ( P " " ) ^ 0 * -1 ) : match ( line ) )
1149     then count = count + 1
1150     end
1151   end
1152   tex.sprint(
1153     luatexbase.catcodetables.expl ,
1154     '\\int_set:Nn \\l_@@_nb_non_empty_lines_int {' .. count .. '}' )
1155 end

1156 \end{luacode*}

```

## 7 History

### Changes between versions 0.6 and 0.7

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

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

### Changes between versions 0.7 and 0.8

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

New key `left-margin`.

### Changes between versions 0.8 and 0.9

New key `tab-size`.

Integer value for the key `splittable`.

## Changes between versions 0.9 and 0.95

New key `show-spaces`.

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

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

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

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

## Changes between versions 0.95 and 0.99

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