

# The package `piton`\*

F. Pantigny  
fpantigny@wanadoo.fr

January 8, 2026

## Abstract

This document is the documented code of the LuaLaTeX package `piton`. It is *not* its user's guide. The guide of utilisation is the document `piton.pdf` (with a French translation: `piton-french.pdf`).

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

## 1 Introduction

The main job of the package `piton` is to take in as input a computer 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 `LPEG1[<language>]` where `<language>` is a Lua string which is the name of the computer language. That LPEG, when matched against the string of a computer 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>1</sup>

In fact, there is a variant of the LPEG `LPEG1[<language>]`, called `LPEG2[<language>]`. The latter uses the first one and will be used to format the whole content of an environment `{Piton}` (with, in particular, small tuning for the beginning and the end).

Consider, for example, the following Python code:

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

The capture returned by the LPEG `LPEG1['python']` (in Lua, this may also be written `LPEG1.python`) against that code is the Lua table containing the following elements :

---

\*This document corresponds to the version 4.11 of `piton`, at the date of 2026/01/08.

<sup>1</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:" }a
{ "{\\PitonStyle{Keyword}{"} }b
{ luatexbase.catcodetables.otherc, "def" }
{ "}}" }
{ luatexbase.catcodetables.other, " " }
{ "{\\PitonStyle{Name.Function}{"} }
{ luatexbase.catcodetables.other, "parity" }
{ "}}" }
{ luatexbase.catcodetables.other, "(" }
{ luatexbase.catcodetables.other, "x" }
{ luatexbase.catcodetables.other, ")" }
{ luatexbase.catcodetables.other, ":" }
{ "\\_piton_end_line: \\_piton_par: \\_piton_begin_line:" }
{ luatexbase.catcodetables.other, " " }
{ "{\\PitonStyle{Keyword}{"} }
{ luatexbase.catcodetables.other, "return" }
{ "}}" }
{ luatexbase.catcodetables.other, " " }
{ luatexbase.catcodetables.other, "x" }
{ "{\\PitonStyle{Operator}{"} }
{ luatexbase.catcodetables.other, "%" }
{ "}}" }
{ "{\\PitonStyle{Number}{"} }
{ luatexbase.catcodetables.other, "2" }
{ "}}" }
{ "\\_piton_end_line:" }

```

<sup>a</sup>Each line of the computer 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 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.other` 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`).

```

\_piton_begin_line:{\\PitonStyle{Keyword}{def}}
{\\PitonStyle{Name.Function}{parity}}(x):\_piton_end_line:\_piton_par:
\_piton_begin_line:~~~~{\\PitonStyle{Keyword}{return}}
{x\\PitonStyle{Operator}{%}}{\\PitonStyle{Number}{2}}\_piton_end_line:

```

## 2 The L3 part of the implementation

### 2.1 Declaration of the package

```

1 < *STY >
2 \NeedsTeXFormat{LaTeX2e}
3 \ProvidesExplPackage
4   {piton}
5   {\PitonFileDate}
6   {\PitonFileVersion}
7   {Highlight computer listings with LPEG on LuaLaTeX}
8 \msg_new:nnn { piton } { latex-too-old }
9   {
10     Your~LaTeX~release-is-too-old. \\

```

```

11   You~need~at~least~the~version~of~2025-06-01.  \\\
12   If~you~use~Overleaf,~you~need~at~least~"TeX-Live-2025".\\
13   The~package~'piton'~won't~be~loaded.
14 }

15 \providecommand { \IfFormatAtLeastTF } { \@ifl@t@r \fmtversion }
16 \IfFormatAtLeastTF
17 { 2025-06-01 }
18 { }
19 { \msg_critical:nn { piton } { latex-too-old } }

```

The command `\text` provided by the package `amstext` will be used to allow the use of the command `\piton{...}` (with the standard syntax) in mathematical mode.

```

20 \RequirePackage { amstext }

```

The command `\marginalia` of the package `marginalia` will be used for the margin notes created by the keys `paperclip` and `annotation`.

```

21 \RequirePackage { marginalia }

```

The package `transparent` is compatible with `pdfmanagement` (which is not loaded by `piton` but which is used for the key `join` when it is loaded).

```

22 \RequirePackage { transparent }

```

```

23 \cs_new_protected:Npn \@@_error:n { \msg_error:nn { piton } }
24 \cs_new_protected:Npn \@@_warning:n { \msg_warning:nn { piton } }
25 \cs_new_protected:Npn \@@_warning:nn { \msg_warning:nnn { piton } }
26 \cs_new_protected:Npn \@@_error:nn { \msg_error:nnn { piton } }
27 \cs_new_protected:Npn \@@_error:nnn { \msg_error:nnnn { piton } }
28 \cs_new_protected:Npn \@@_fatal:n { \msg_fatal:nn { piton } }
29 \cs_new_protected:Npn \@@_fatal:nn { \msg_fatal:nnn { piton } }
30 \cs_new_protected:Npn \@@_msg_new:nn { \msg_new:nnn { piton } }

```

With Overleaf (and also TeXPage), by default, a document is compiled in non-stop mode. When there is an error, there is no way to the user to use the key `H` in order to have more information. That's why we decide to put that piece of information (for the messages with such information) in the main part of the message when the key `messages-for-Overleaf` is used (at load-time).

```

31 \cs_new_protected:Npn \@@_msg_new:nnn #1 #2 #3
32 {
33   \bool_if:NTF \g_@@_messages_for_Overleaf_bool
34     { \msg_new:nnn { piton } { #1 } { #2 } { #3 } }
35     { \msg_new:nnnn { piton } { #1 } { #2 } { #3 } }
36 }

```

We also create commands which will generate usually an error but only a warning on Overleaf. The argument is given by curryfication.

```

37 \cs_new_protected:Npn \@@_error_or_warning:n
38 { \bool_if:NTF \g_@@_messages_for_Overleaf_bool \@@_warning:n \@@_error:n }
39 \cs_new_protected:Npn \@@_error_or_warning:nn
40 { \bool_if:NTF \g_@@_messages_for_Overleaf_bool \@@_warning:nn \@@_error:nn }

```

We try to detect whether the compilation is done on Overleaf. We use `\c_sys_jobname_str` because, with Overleaf, the value of `\c_sys_jobname_str` is always “output”.

```

41 \bool_new:N \g_@@_messages_for_Overleaf_bool
42 \bool_gset:Nn \g_@@_messages_for_Overleaf_bool
43 {
44   \str_if_eq_p:on \c_sys_jobname_str { _region_ } % for Emacs
45   || \str_if_eq_p:on \c_sys_jobname_str { output } % for Overleaf
46 }

```

```

47 \@@_msg_new:nn { LuaLaTeX-mandatory }
48 {

```

```

49   LuaLaTeX-is-mandatory.\\
50   The-package~'piton'-requires-the-engine-LuaLaTeX.\\
51   \str_if_eq:onT \c_sys_jobname_str { output }
52   { If~you-use-Overleaf,~you-can-switch-to-LuaLaTeX-in~
53     "Settings->~Compiler"~and-if-you-use-TeXPage,
54     ~you-should-go-in~"Settings". \\ }
55   \IfClassLoadedT { beamer }
56   {
57     Since-you-use-Beamer,~don't~forget~to~use~piton~in~frames~with~
58     the~key~'fragile'.\\
59   }
60   \IfClassLoadedT { ltx-talk }
61   {
62     Since-you-use~'ltx-talk',~don't~forget~to~use~piton~in~
63     environments~'frame*'.\\
64   }
65   That~error~is~fatal.
66 }
67 \sys_if_engine_luatex:F { \@@_fatal:n { LuaLaTeX-mandatory } }

68 \RequirePackage { luacode }

69 \@@_msg_new:nnn { piton.lua-not-found }
70 {
71   The~file~'piton.lua'~can't~be~found.\\
72   This~error~is~fatal.\\
73   If~you~want~to~know~how~to~retrieve~the~file~'piton.lua',~type-H~<return>.
74 }
75 {
76   On~the~site~CTAN,~go~to~the~page~of~'piton':~https://ctan.org/pkg/piton.~
77   The~file~'README.md'~explains~how~to~retrieve~the~files~'piton.sty'~and~
78   'piton.lua'.
79 }

80 \file_if_exist:nF { piton.lua } { \@@_fatal:n { piton.lua-not-found } }

```

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

```
81 \bool_new:N \g_@@_footnotehyper_bool
```

The boolean `\g_@@_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.

```
82 \bool_new:N \g_@@_footnote_bool
```

```
83 \bool_new:N \g_@@_beamer_bool
```

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

```

84 \keys_define:nn { piton }
85 {
86   footnote .bool_gset:N = \g_@@_footnote_bool ,
87   footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
88   footnote .usage:n = load ,
89   footnotehyper .usage:n = load ,
90
91   beamer .bool_gset:N = \g_@@_beamer_bool ,
92   beamer .default:n = true ,
93   beamer .usage:n = load ,
94
95   unknown .code:n = \@@_error:n { Unknown-key-for-package }
96 }
97 \@@_msg_new:nn { Unknown-key-for-package }
98 {

```

```

99   Unknown~key.\\
100   You~have~used~the~key~'\l_keys_key_str'~when~loading~piton~
101   but~the~only~keys~available~here~are~'beamer',~'footnote'~
102   and~'footnotehyper'.~Other~keys~are~available~in~
103   \token_to_str:N \PitonOptions.\\
104   That~key~will~be~ignored.
105 }

```

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

```

106 \ProcessKeyOptions

107 \IfClassLoadedT { beamer } { \bool_gset_true:N \g_@@_beamer_bool }
108 \IfClassLoadedT { ltx-talk } { \bool_gset_true:N \g_@@_beamer_bool }
109 \IfPackageLoadedT { beamerarticle } { \bool_gset_true:N \g_@@_beamer_bool }

110 \lua_now:e
111 {
112   piton = piton~or~{ }
113   piton.last_code = ''
114   piton.last_language = ''
115   piton.join = ''
116   piton.write = ''
117   piton.path_write = ''
118   \bool_if:NT \g_@@_beamer_bool { piton.beamer = true }
119 }

120 \RequirePackage { xcolor }

121 \@@_msg_new:nn { footnote~with~footnotehyper~package }
122 {
123   Footnote~forbidden.\\
124   You~can't~use~the~option~'footnote'~because~the~package~
125   footnotehyper~has~already~been~loaded.~
126   If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
127   within~the~environments~of~piton~will~be~extracted~with~the~tools~
128   of~the~package~footnotehyper.\\
129   If~you~go~on,~the~package~footnote~won't~be~loaded.
130 }

131 \@@_msg_new:nn { footnotehyper~with~footnote~package }
132 {
133   You~can't~use~the~option~'footnotehyper'~because~the~package~
134   footnote~has~already~been~loaded.~
135   If~you~want,~you~can~use~the~option~'footnote'~and~the~footnotes~
136   within~the~environments~of~piton~will~be~extracted~with~the~tools~
137   of~the~package~footnote.\\
138   If~you~go~on,~the~package~footnotehyper~won't~be~loaded.
139 }

140 \bool_if:NT \g_@@_footnote_bool
141 {

```

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

```

142   \IfClassLoadedTF { beamer }
143   { \bool_gset_false:N \g_@@_footnote_bool }
144   {
145     \IfPackageLoadedTF { footnotehyper }
146     { \@@_error:n { footnote~with~footnotehyper~package } }
147     { \usepackage { footnote } }
148   }
149 }

150 \bool_if:NT \g_@@_footnotehyper_bool
151 {

```

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

```

152 \IfClassLoadedTF { beamer }
153 { \bool_gset_false:N \g_@@_footnote_bool }
154 {
155   \IfPackageLoadedTF { footnote }
156   { \@@_error:n { footnotehyper~with~footnote~package } }
157   { \usepackage { footnotehyper } }
158   \bool_gset_true:N \g_@@_footnote_bool
159 }
160 }

```

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

## 2.2 Parameters and technical definitions

```

161 \dim_new:N \l_@@_rounded_corners_dim
162 \bool_new:N \l_@@_in_label_bool
163 \dim_new:N \l_@@_tmpc_dim

```

The listing that we have to format will be stored in `\l_@@_listing_tl`. That applies both for the command `\PitonInputFile` and the environment `{Piton}` (or another environment defined by `\NewPitonEnvironment`).

```

164 \tl_new:N \l_@@_listing_tl

```

The content of an environment such as `{Piton}` will be composed first in the following box, but that box will (sometimes) be *unboxed* at the end.

We need a global variable (see `\@@_add_bg_and_right_nb_to_output_box:`).

```

165 \box_new:N \g_@@_output_box

```

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

```

166 \str_new:N \l_piton_language_str
167 \str_set:Nn \l_piton_language_str { python }

```

Each time an environment of `piton` is used, the computer listing in the body of that environment will be stored in the following global string.

```

168 \tl_new:N \g_piton_last_code_tl

```

The following parameter corresponds to the key `path` (which is the path used to include files by `\PitonInputFile`). Each component of that sequence will be a string (type `str`).

```

169 \seq_new:N \l_@@_path_seq

```

The names of all the join files will be stored in the following sequence:

```

170 \seq_new:N \g_@@_join_seq

```

The following parameter corresponds to the key `path-write` (which is the path used when writing files from listings inserted in the environments of `piton` by use of the key `write`).

```

171 \str_new:N \l_@@_path_write_str

```

The following parameter corresponds to the key `tcolorbox`.

```

172 \bool_new:N \l_@@_tcolorbox_bool

```

When the key `tcolorbox` is used, you will have to take into account the width of the graphical elements added by `tcolorbox` on both sides of the listing. We will put that quantity in the following variable.

```

173 \dim_new:N \l_@@_tcb_margins_dim

```

The following parameter corresponds to the key `box`.

```

174 \str_new:N \l_@@_box_str

```

In order to have a better control over the keys.

```

175 \bool_new:N \l_@@_in_PitonOptions_bool

```

```
176 \bool_new:N \l_@@_in_PitonInputFile_bool
```

The following parameter corresponds to the key `font-command`.

```
177 \tl_new:N \l_@@_font_command_tl
178 \tl_set:Nn \l_@@_font_command_tl { \ttfamily }
```

We will compute (with Lua) the numbers of lines of the listings (or *chunks* of listings when `split-on-empty-lines` is in force) and store it in the following counter.

```
179 \int_new:N \g_@@_nb_lines_int
```

The same for the number of non-empty lines of the listings.

```
180 \int_new:N \l_@@_nb_non_empty_lines_int
```

The following counter will be used to count the lines during the composition. It will take into account all the lines, empty or not empty. It won't be used to print the numbers of the lines but will be used to allow or disallow line breaks (when `splittable` is in force) and for the color of the background (when `background-color` is used with a *list* of colors or when `\rowcolor` is used).

```
181 \int_new:N \g_@@_line_int
```

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 a listing (or a *chunk* of listings when the key `split-on-empty-lines` is in force).

```
182 \int_new:N \l_@@_splittable_int
```

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

```
183 \int_set:Nn \l_@@_splittable_int { 100 }
```

When the key `split-on-empty-lines` will be in force, then the following token list will be inserted between the chunks of code (the computer listing provided by the end user is split in chunks on the empty lines in the code).

```
184 \tl_new:N \l_@@_split_separation_tl
185 \tl_set:Nn \l_@@_split_separation_tl
186 { \vspace { \baselineskip } \vspace { -1.25pt } }
```

That parameter must contain elements to be inserted in *vertical* mode by TeX.

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

```
187 \clist_new:N \l_@@_bg_color_clist
```

We will also keep in memory the length of the previous `clist` (for efficiency).

```
188 \int_new:N \l_@@_bg_colors_int
```

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

```
189 \tl_new:N \l_@@_prompt_bg_color_tl
190 \tl_set:Nn \l_@@_prompt_bg_color_tl { gray!15 }
```

```
191 \tl_new:N \l_@@_space_in_string_tl
```

The following parameters correspond to the keys `begin-range` and `end-range` of the command `\PitonInputFile`.

```
192 \str_new:N \l_@@_begin_range_str
193 \str_new:N \l_@@_end_range_str
```

The following boolean corresponds to the key `math-comments` (available only in the preamble of the LaTeX document).

```
194 \bool_new:N \g_@@_math_comments_bool
```

The argument of `\PitonInputFile`.

```
195 \str_new:N \l_@@_file_name_str
```

The following flag corresponds to the key `print`. The initial value of that parameter will be `true` (and not `false`) since, of course, by default, we want to print the content of the environment `{Piton}`

```

196 \bool_new:N \l_@@_print_bool
197 \bool_set_true:N \l_@@_print_bool

```

The parameter `\l_@@_write_str` corresponds to the key `write`.

```

198 \str_new:N \l_@@_write_str

```

The parameter `\l_@@_join_str` corresponds to the key `join`.

```

199 \str_new:N \l_@@_join_str
200 \str_new:N \l_@@_join_separation_str
201 \str_set:Nn \l_@@_join_separation_str { }

```

The following boolean corresponds to the keys `paperclip` and `annotation`.

```

202 \bool_new:N \l_@@_paperclip_bool
203 \str_new:N \l_@@_paperclip_str
204 \bool_new:N \l_@@_annotation_bool

```

The listings embedded in the PDF by the key `paperclip` will be numbered by the following counter.

```

205 \int_new:N \g_@@_paperclip_int

```

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

```

206 \bool_new:N \l_@@_show_spaces_bool

```

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

```

207 \bool_new:N \l_@@_break_lines_in_Piton_bool
208 \bool_set_true:N \l_@@_break_lines_in_Piton_bool
209 \bool_new:N \l_@@_indent_broken_lines_bool

```

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

```

210 \tl_new:N \l_@@_continuation_symbol_tl
211 \tl_set:Nn \l_@@_continuation_symbol_tl { + }

```

The following token list corresponds to the key `continuation-symbol-on-indentation`. The name has been shorten to `csoi`.

```

212 \tl_new:N \l_@@_csoi_tl
213 \tl_set:Nn \l_@@_csoi_tl { $ \hookrightarrow \; $ }

```

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

```

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

```

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

```

216 \bool_new:N \l_@@_break_lines_in_piton_bool

```

The following flag will be raised when the key `max-width` is used (and when `width` is used with the key `min`, which is equivalent to `max-width=\linewidth`). Note also that the key `box` sets `width=min` (except if `min` is used with a numerical value).

```

217 \bool_new:N \l_@@_minimize_width_bool

```

The following dimension corresponds to the key `width`. It's meant to be the whole width of the environment (for instance, the width of the box of `tcolorbox` when the key `tcolorbox` is used). The initial value is 0 pt which means that the end user has not used the key. In that case, it will be set equal to the current value of `\linewidth` in `\@@_pre_composition:`.

However if `max-width` is used (or `width=min` which is equivalent to `max-width=\linewidth`), the actual width of the final environment in the PDF may (potentially) be smaller.

```

218 \dim_new:N \l_@@_width_dim

```

`\l_@@_listing_width_dim` will be the width of the listing taking into account the lines of code (of course) but also:

- `\l_@@_left_margin_dim` (for the numbers of lines);
- a small margin when `background-color` is in force<sup>2</sup>).

---

<sup>2</sup>Remark that the mere use of `\rowcolor` does not add those small margins.



```
219 \dim_new:N \l_@@_listing_width_dim
```

However, if `max-width` is used (or `width=min` which is equivalent to `max-width=\linewidth`), that length will be computed once again in `\@@_create_output_box`:

`\l_@@_code_width_dim` will be the length of the lines of code, without the potential margins (for the backgrounds and for `length-margin` for the number of lines).

It will be computed in `\@@_compute_code_width`:

```
220 \dim_new:N \l_@@_code_width_dim
```

```
221 \box_new:N \l_@@_line_box
```

The following dimension corresponds to the keys `left-margin` and `right-margin`.

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

```
223 \dim_new:N \l_@@_right_margin_dim
```

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

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

```
225 \bool_new:N \l_@@_right_margin_auto_bool
```

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

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

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

The following parameter corresponds to the key `numbers/step`.

```
228 \int_new:N \l_@@_numbers_step_int
```

```
229 \int_set:Nn \l_@@_numbers_step_int { 1 }
```

When the key `line-numbers/position` is set to `right`, we will have to keep in memory the numbers of the lines in the following sequence.

```
230 \seq_new:N \g_@@_visual_line_numbers_seq
```

Be careful. The following sequence `\g_@@_languages_seq` is not the list of the languages supported by `piton`. It's the list of the languages for which at least a user function has been defined. We need that sequence only for the command `\PitonClearUserFunctions` when it is used without its optional argument: it must clear the whole list of languages for which at least a user function has been defined.

```
231 \seq_new:N \g_@@_languages_seq
```

```
232 \int_new:N \l_@@_tab_size_int
```

```
233 \int_set:Nn \l_@@_tab_size_int { 4 }
```

```
234 \cs_new_protected:Npn \@@_tab:
```

```
235 {
```

```
236   \bool_if:NTF \l_@@_show_spaces_bool
```

```
237   {
```

```
238     \hbox_set:Nn \l_tmpa_box
```

```
239     { \prg_replicate:nn \l_@@_tab_size_int { ~ } }
```

```
240     \dim_set:Nn \l_tmpa_dim { \box_wd:N \l_tmpa_box }
```

```
241     \< \mathcolor { gray }
```

```
242     { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \>
```

```
243   }
```

```
244   { \hbox:n { \prg_replicate:nn \l_@@_tab_size_int { ~ } } }
```

```
245   \int_gadd:Nn \g_@@_indentation_int \l_@@_tab_size_int
```

```
246 }
```

The following integer corresponds to the key `gobble`.

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

The following token list will be used only for the spaces in the strings.

```
248 \tl_set_eq:NN \l_@@_space_in_string_tl \nobreakspace
```

When the key `break-lines-in-piton` is set, that parameter will be replaced by `\space` (in `\piton` with the standard syntax) and when the key `show-spaces-in-strings` is set, it will be replaced by `\U+2423`.

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

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

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

```
250 \cs_new_protected:Npn \@@_label:n #1
251 {
252   \bool_if:NTF \l_@@_line_numbers_bool
253   {
254     \@bsphack
255     \protected@write \@auxout { }
256     {
257       \string \newlabel { #1 }
258       {
259         { \int_use:N \g_@@_visual_line_int }
260         { \thepage }
261         { }
262         { line.#1 }
263         { }
264       }
265     }
266     \@esphack
267     \IfPackageLoadedT { hyperref }
268     { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
269   }
270   { \@@_error:n { label-with-lines-numbers } }
271 }
```

The same goes for the command `\zlabel` if the `zref` package is loaded. Note that `\label` will also be linked to `\@@_zlabel:n` if the key `label-as-zlabel` is set to `true`.

```
272 \cs_new_protected:Npn \@@_zlabel:n #1
273 {
274   \bool_if:NTF \l_@@_line_numbers_bool
275   {
276     \@bsphack
277     \protected@write \@auxout { }
278     {
279       \string \zref@newlabel { #1 }
280       {
281         \string \default { \int_use:N \g_@@_visual_line_int }
282         \string \page { \thepage }
283         \string \zc@type { line }
284         \string \anchor { line.#1 }
285       }
286     }
287     \@esphack
288     \IfPackageLoadedT { hyperref }
289     { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
290   }
291   { \@@_error:n { label-with-lines-numbers } }
292 }
```

In the environments `{Piton}` the command `\rowcolor` will be linked to the following one.

```
293 \NewDocumentCommand { \@@_rowcolor:n } { o m }
294 {
295   \tl_gset:ce
296   { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 }_ t1 }
297   { \tl_if_novalue:nTF { #1 } { #2 } { [ #1 ] { #2 } } }
298   \bool_gset_true:N \g_@@_rowcolor_inside_bool
299 }
```

In the command `piton` (in fact in `\@@_piton_standard` and `\@@_piton_verbatim`, the command `\rowcolor` will be linked to the following one (in order to nullify its effect).

```
300 \NewDocumentCommand { \@@_noop_rowcolor } { o m } { }
```

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

These macros must *not* be protected.

```
301 \cs_new:Npn \@@_marker_beginning:n #1 { }
302 \cs_new:Npn \@@_marker_end:n #1 { }
```

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

```
303 \tl_new:N \g_@@_after_line_tl
```

The spaces at the end of a line of code are deleted by `piton`. However, it’s not actually true: they are replace by `\@@_trailing_space:`.

```
304 \cs_new_protected:Npn \@@_trailing_space: { }
```

When we have to rescan some pieces of code, we will use `\@@_piton:n` and that command `\@@_piton:n` will set `\@@_trailing_space:` equal to `\space`.

```
305 \bool_new:N \g_@@_color_is_none_bool
306 \bool_new:N \g_@@_next_color_is_none_bool
```

```
307 \bool_new:N \g_@@_rowcolor_inside_bool
```

## 2.3 Detected commands

There are four keys for “detected commands and environments”: `detected-commands`, `raw-detected-commands`, `beamer-commands` and `beamer-environments`.

In fact, there is also `vertical-detected-commands` but has a special treatment.

For each of those keys, we keep a clist of the names of such detected commands and environments. For the commands, the corresponding clist will contain the name of the commands *without* the backlash.

```
308 \clist_new:N \l_@@_detected_commands_clist
309 \clist_new:N \l_@@_raw_detected_commands_clist
310 \clist_new:N \l_@@_beamer_commands_clist
311 \clist_set:Nn \l_@@_beamer_commands_clist
312   { uncover , only , visible , invisible , alert , action }
313 \clist_new:N \l_@@_beamer_environments_clist
314 \clist_set:Nn \l_@@_beamer_environments_clist
315   { uncoverenv , onlyenv , visibleenv , invisibleenv , alertenv , actionenv }
```

Remark that, since we have used clists, these clists, as token lists are “purified”: there is no empty component and for each component, there is no space on both sides.

Of course, the value of those clists may be modified during the preamble of the document by using the corresponding key (`detected-commands`, etc.).

However, after the `\begin{document}`, it’s no longer possible to modify those clists because their contents will be used in the construction of the main LPEG for each computer language.

However, in a `\AtBeginDocument`, we will convert those clists into “toks registers” of TeX.

```
316 \hook_gput_code:nnn { begindocument } { . }
317   {
```

```

318 \newtoks \PitonDetectedCommands
319 \newtoks \PitonRawDetectedCommands
320 \newtoks \PitonBeamerCommands
321 \newtoks \PitonBeamerEnvironments

```

L3 does *not* support those “toks registers” but it’s still possible to affect to the “toks registers” the content of the clists with a L3-like syntax.

```

322 \exp_args:NV \PitonDetectedCommands \l_@@_detected_commands_clist
323 \exp_args:NV \PitonRawDetectedCommands \l_@@_raw_detected_commands_clist
324 \exp_args:NV \PitonBeamerCommands \l_@@_beamer_commands_clist
325 \exp_args:NV \PitonBeamerEnvironments \l_@@_beamer_environments_clist
326 }

```

Then at the beginning of the document, when we will load the Lua file `piton.lua`, we will read those “toks registers” within Lua (with `tex.toks`) and convert them into Lua tables (and, then, use those tables to construct LPEG).

When the key `vertical-detected-commands` is used, we will have to redefine the corresponding commands in `\@@_pre_composition:`.

The instructions for these redefinitions will be put in the following token list.

```

327 \tl_new:N \g_@@_def_vertical_commands_tl

328 \cs_new_protected:Npn \@@_vertical_commands:n #1
329 {
330   \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 }
331   \clist_map_inline:nn { #1 }
332   {
333     \cs_set_eq:cc { @@_old_ ##1 : } { ##1 }
334     \cs_new_protected:cn { @@_new_ ##1 : n }
335     {
336       \bool_if:nTF
337       { \l_@@_tcolorbox_bool || ! \str_if_empty_p:N \l_@@_box_str }
338       {
339         \tl_gput_right:Nn \g_@@_after_line_tl
340         { \use:c { @@_old_ ##1 : } { ####1 } }
341       }
342       {
343         \cs_if_exist:cTF { g_@@_after_line _ \int_use:N \g_@@_line_int _ tl }
344         { \tl_gput_right:cn }
345         { \tl_gset:cn }
346         { g_@@_after_line _ \int_eval:n { \g_@@_line_int + 1 } _ tl }
347         { \use:c { @@_old_ ##1 : } { ####1 } }
348       }
349     }
350     \tl_gput_right:Nn \g_@@_def_vertical_commands_tl
351     { \cs_set_eq:cc { ##1 } { @@_new_ ##1 : n } }
352   }
353 }

```

## 2.4 Treatment of a line of code

```

354 \cs_new_protected:Npn \@@_replace_spaces:n #1
355 {
356   \tl_set:Nn \l_tmpa_tl { #1 }
357   \bool_if:NTF \l_@@_show_spaces_bool
358   {
359     \tl_set:Nn \l_@@_space_in_string_tl { } % U+2423
360     \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl { } % U+2423
361   }
362   {

```

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

```

363 \bool_if:NT \l_@@_break_lines_in_Piton_bool
364 {
365     \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
366     { \tl_set_eq:NN \l_@@_space_in_string_tl \@@_breakable_space: }

```

In the following code, we have to replace all the spaces in the token list `\l_tmpa_tl`. That means that this replacement must be “recursive”: even the spaces which are within brace groups (`{...}`) must be replaced. For instance, the spaces in long strings of Python are within such groups since there are within a command `\PitonStyle{String.Long}{...}`. That’s why the use of `\tl_replace_all:Nnn` is not enough.

The first implementation was using `\tl_regex_replace_all:nnN`

```
\tl_regex_replace_all:nnN { \x20 } { \c { @@_breakable_space: } } \l_tmpa_tl
```

but that programming was certainly slow.

Now, we use `\tl_replace_all:NvN` *but*, in the styles `String.Long.Internal` we replace the spaces with `\@@_breakable_space:` by another use of the same technic with `\tl_replace_all:NvN`. We do the same jog for the *doc strings* of Python and for the comments.

```

367 \tl_replace_all:NvN \l_tmpa_tl
368 \c_catcode_other_space_tl
369 \@@_breakable_space:
370 }
371 }
372 \l_tmpa_tl
373 }
374 \cs_generate_variant:Nn \@@_replace_spaces:n { o }

```

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

`\@@_begin_line:` is a TeX command with a delimited argument (`\@@_end_line:` is the marker for the end of the argument).

However, we define also `\@@_end_line:` as no-op, because, when the last line of the listing is the end of an environment of Beamer (eg `\end{uncoverenv}`), we will have a token `\@@_end_line:` added at the end without any corresponding `\@@_begin_line:`).

```

375 \cs_set_protected:Npn \@@_end_line: { }

376 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
377 {
378     \group_begin:
379     \int_gzero:N \g_@@_indentation_int

```

We put the potential number of line, the potential left and right margins.

```

380 \hbox_set:Nn \l_@@_line_box
381 {
382     \skip_horizontal:N \l_@@_left_margin_dim
383     \bool_if:NT \l_@@_line_numbers_bool
384     {

```

`\l_tmpa_int` will be equal to 1 when the current line is not empty.

```

385 \int_set:Nn \l_tmpa_int
386 {
387     \lua_now:e
388     {
389         tex.sprint
390         (

```

The following expression gives an integer of Lua (*integer* is a sub-type of *number* introduced in Lua 5.3), the output will be of the form “3” (and not “3.0”) which is what we want for `\int_set:Nn`.

```

391         piton.empty_lines
392         [ \int_eval:n { \g_@@_line_int + 1 } ]
393     )

```

```

394     }
395   }
396   \bool_lazy_or:nnT
397   { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
398   { ! \l_@@_skip_empty_lines_bool }
399   { \int_gincr:N \g_@@_visual_line_int }
400
401   \bool_lazy_or:nnTF
402   { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
403   { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
404   {
405     \bool_lazy_or:nnTF
406     { \int_compare_p:nNn { \l_@@_numbers_step_int } = 1 }
407     {
408       \int_compare_p:nNn
409       {
410         \int_mod:nn
411         { \g_@@_visual_line_int }
412         { \l_@@_numbers_step_int }
413       }
414       = \c_one_int
415     }
416     {
417       \str_if_eq:eeTF \l_@@_line_numbers_position_str { left }
418       { \@@_print_number_left: }
419       {
420         \seq_gput_right:Ne \g_@@_visual_line_numbers_seq
421         { \int_use:N \g_@@_visual_line_int }
422       }
423     }
424     {
425       \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
426       { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
427     }
428   }
429   {
430     \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
431     { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
432   }
433 }

```

If there is a background, we must remind that there is a left margin of 0.5 em for the background (which will be added later).

```

434   \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
435   {
... but if only if the key left-margin is not used !
436     \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
437     { \skip_horizontal:n { 0.5 em } }
438   }

```

```

439   \bool_if:NTF \l_@@_minimize_width_bool
440   {
441     \hbox_set:Nn \l_tmpa_box
442     {
443       \language = -1
444       \raggedright
445       \strut
446       \@@_replace_spaces:n { #1 }
447       \strut \hfil
448     }
449     \dim_compare:nNnTF { \box_wd:N \l_tmpa_box } < \l_@@_code_width_dim
450     { \box_use:N \l_tmpa_box }
451     { \@@_vtop_of_code:n { #1 } }
452   }

```

```

453         { \@@_vtop_of_code:n { #1 } }
454     }

```

Now, the line of code is composed in the box `\l_@@_line_box`.

```

455     \box_set_dp:Nn \l_@@_line_box { \box_dp:N \l_@@_line_box + 1.25 pt }
456     \box_set_ht:Nn \l_@@_line_box { \box_ht:N \l_@@_line_box + 1.25 pt }
457     \box_use_drop:N \l_@@_line_box
458     \group_end:
459     \g_@@_after_line_tl
460     \tl_gclear:N \g_@@_after_line_tl
461 }

```

The following command will be used in `\@@_begin_line: ... \@@_end_line:.`

```

462 \cs_new_protected:Npn \@@_vtop_of_code:n #1
463 {
464     \vbox_top:n
465     {
466         \hspace = \l_@@_code_width_dim
467         \language = -1
468         \raggedright
469         \strut
470         \@@_replace_spaces:n { #1 }
471         \strut \hfil
472     }
473 }

```

The following command will be used when the key `background-color` is used or when the key `line-numbers` is used in conjunction with `line-numbers/position=right`.

The content of the line has been previously set in `\l_@@_line_box`.

That command is used only once, in `\@@_add_bg_and_right_nb_to_output_box:.`

```

474 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_line_and_use:
475 {
476     \vtop
477     {
478         \offinterlineskip
479         \hbox
480         {

```

The command `\@@_compute_and_set_color:` sets the current color but also sets the booleans `\g_@@_color_is_none_bool` and `\g_@@_next_color_is_none_bool`. It uses the current value of `\l_@@_bg_color_clist`, the value of `\g_@@_line_int` (the number of the current line) but also potential token lists of the form `\g_@@_color_12_tl` if the end user has used the command `\rowcolor`.

```

481         \group_begin:
482         \@@_compute_and_set_color:

```

The colored panels are overlapping. However, if the special color `none` is used we must not put such overlapping.

```

483         \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
484         \bool_if:NT \g_@@_next_color_is_none_bool
485         { \dim_sub:Nn \l_tmpa_dim { 2.5 pt } }

```

When `\g_@@_color_is_none_bool` is in force, we will compose a `\vrule` of width 0 pt. We need that `\vrule` because it will be a strut.

```

486         \bool_if:NTF \g_@@_color_is_none_bool
487         { \dim_zero:N \l_tmpb_dim }
488         { \dim_set_eq:NN \l_tmpb_dim \l_@@_listing_width_dim }
489         \dim_set:Nn \l_@@_tmpc_dim { \box_ht:N \l_@@_line_box }

```

Now, the colored panel.

```

490         \dim_compare:nNnTF \l_@@_rounded_corners_dim > \c_zero_dim
491         {
492             \int_compare:nNnTF \g_@@_line_int = \c_one_int
493             {

```

```

494 \begin{tikzpicture}[baseline = 0cm]
495 \fill (0,0)
496 [rounded-corners = \l_@@_rounded_corners_dim]
497 -- (0,\l_@@_tmpc_dim)
498 -- (\l_tmpb_dim,\l_@@_tmpc_dim)
499 [sharp-corners] -- (\l_tmpb_dim,-\l_tmpa_dim)
500 -- (0,-\l_tmpa_dim)
501 -- cycle ;
502 \end{tikzpicture}
503 }
504 {
505 \int_compare:nNnTF \g_@@_line_int = \g_@@_nb_lines_int
506 {
507 \begin{tikzpicture}[baseline = 0cm]
508 \fill (0,0) -- (0,\l_@@_tmpc_dim)
509 -- (\l_tmpb_dim,\l_@@_tmpc_dim)
510 [rounded-corners = \l_@@_rounded_corners_dim]
511 -- (\l_tmpb_dim,-\l_tmpa_dim)
512 -- (0,-\l_tmpa_dim)
513 -- cycle ;
514 \end{tikzpicture}
515 }
516 {
517 \vrule height \l_@@_tmpc_dim
518 depth \l_tmpa_dim
519 width \l_tmpb_dim

```

For the case when line-numbers/position=right is in force with line-numbers.

```

520 % added 2026-01-02
521 \dim_compare:nNnT \l_tmpb_dim = \c_zero_dim
522 { \skip_horizontal:N \l_@@_listing_width_dim }
523 }
524 }
525 }
526 {
527 \vrule height \l_@@_tmpc_dim
528 depth \l_tmpa_dim
529 width \l_tmpb_dim

```

For the case when line-numbers/position=right is in force with line-numbers.

```

530 % added 2026-01-02
531 \dim_compare:nNnT \l_tmpb_dim = \c_zero_dim
532 { \skip_horizontal:N \l_@@_listing_width_dim }
533 }

```

The group is for the color of the background.

```

534 \group_end:
535 % added 2026-01-02
536 \bool_if:NT \l_@@_line_numbers_bool
537 {
538 \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
539 {
540 \seq_gpop_right:NN \g_@@_visual_line_numbers_seq \l_tmpa_tl
541 \@@_print_number_right:
542 }
543 }
544 }
545 \bool_if:NT \g_@@_next_color_is_none_bool
546 { \skip_vertical:n { 2.5 pt } }
547 \skip_vertical:n { - \box_ht_plus_dp:N \l_@@_line_box }
548 \box_use_drop:N \l_@@_line_box
549 }
550 }

```

End of \@@\_add\_bg\_and\_right\_nb\_to\_line\_and\_use:



The command `\@@_compute_and_set_color:` sets the current color but also sets the booleans `\g_@@_color_is_none_bool` and `\g_@@_next_color_is_none_bool`. It uses the current value of `\l_@@_bg_color_clist`, the value of `\g_@@_line_int` (the number of the current line) but also potential token lists of the form `\g_@@_color_12_tl` if the end user has used the command `\rowcolor`.

```

551 \cs_set_protected:Npn \@@_compute_and_set_color:
552 {
553   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
554     { \tl_set:Nn \l_tmpa_tl { none } }
555     {
556       \int_set:Nn \l_tmpb_int
557         { \int_mod:nn \g_@@_line_int \l_@@_bg_colors_int + 1 }
558       \tl_set:Ne \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
559     }

```

The row may have a color specified by the command `\rowcolor`. We check that point now.

```

560 \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
561 {
562   \tl_set_eq:Nc \l_tmpa_tl { g_@@_color_ \int_use:N \g_@@_line_int _ tl }

```

We don't need any longer the variable and that's why we delete it (it must be free for the next environment of `piton`).

```

563   \cs_undefine:c { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
564 }
565 \tl_if_eq:NnTF \l_tmpa_tl { none }
566 { \bool_gset_true:N \g_@@_color_is_none_bool }
567 {
568   \bool_gset_false:N \g_@@_color_is_none_bool
569   \@@_color:o \l_tmpa_tl
570 }

```

We are looking for the next color because we have to know whether that color is the special color `none` (for the vertical adjustment of the background color).

```

571 \int_compare:nNnTF { \g_@@_line_int + 1 } = \g_@@_nb_lines_int
572 { \bool_gset_false:N \g_@@_next_color_is_none_bool }
573 {
574   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
575     { \tl_set:Nn \l_tmpa_tl { none } }
576     {
577       \int_set:Nn \l_tmpb_int
578         { \int_mod:nn { \g_@@_line_int + 1 } \l_@@_bg_colors_int + 1 }
579       \tl_set:Ne \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
580     }
581   \cs_if_exist:cT { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
582   {
583     \tl_set_eq:Nc \l_tmpa_tl
584       { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
585   }
586   \tl_if_eq:NnTF \l_tmpa_tl { none }
587   { \bool_gset_true:N \g_@@_next_color_is_none_bool }
588   { \bool_gset_false:N \g_@@_next_color_is_none_bool }
589 }
590 }

```

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

```

591 \cs_set_protected:Npn \@@_color:n #1
592 {
593   \tl_if_head_eq_meaning:nNTF { #1 } [
594     {
595       \tl_set:Nn \l_tmpa_tl { #1 }
596       \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
597       \exp_last_unbraced:No \color \l_tmpa_tl
598     }
599     { \color { #1 } }
600   }
601 \cs_generate_variant:Nn \@@_color:n { o }

```

The command `\@@_par:` will be inserted by Lua between two lines of the computer listing.

- In fact, it will be inserted between two commands `\@@_begin_line:...``\@@_end_of_line:.`
- When the key `break-lines-in-Piton` is in force, a line of the computer listing (the *input*) may result in several lines in the PDF (the *output*).
- Remind that `\@@_par:` has a rather complex behaviour because it will finish and start paragraphs.

```
602 \cs_new_protected:Npn \@@_par:
603 {
```

We recall that `\g_@@_line_int` is *not* used for the number of line printed in the PDF (when `line-numbers` is in force)...

```
604 \int_gincr:N \g_@@_line_int
```

... it will be used to allow or disallow page breaks, and also by the command `\rowcolor`.

Each line in the listing is composed in a box of TeX (which may contain several lines when the key `break-lines-in-Piton` is in force) put in a paragraph.

```
605 \par
```

We now add a `\kern` because each line of code is overlapping vertically by a quantity of 2.5 pt in order to have a good background (when `background-color` is in force). We need to use a `\kern` (in fact `\par\kern...`) and not a `\vskip` because page breaks should *not* be allowed on that kern.

```
606 \kern -2.5 pt
```

Now, we control page breaks after the paragraph.

```
607 \@@_add_penalty_for_the_line:
608 }
```

After the command `\@@_par:`, we will usually have a command `\@@_begin_line:.`

The following command `\@@_breakable_space:` is for breakable spaces in the environments `{Piton}` and the listings of `\PitonInputFile` and *not* for the commands `\piton`.

```
609 \cs_set_protected:Npn \@@_breakable_space:
610 {
611   \discretionary
612     { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
613     {
614       \hbox_overlap_left:n
615         {
616           {
617             \normalfont \footnotesize \color { gray }
618             \l_@@_continuation_symbol_tl
619           }
620           \skip_horizontal:n { 0.3 em }
621           \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
622             { \skip_horizontal:n { 0.5 em } }
623         }
624       \bool_if:NT \l_@@_indent_broken_lines_bool
625       {
626         \hbox:n
627           {
628             \prg_replicate:nn { \g_@@_indentation_int } { ~ }
629             { \color { gray } \l_@@_csoi_tl }
630           }
631       }
632     }
633   { \hbox { ~ } }
634 }
```

## 2.5 PitonOptions

```
635 \bool_new:N \l_@@_line_numbers_bool
636 \bool_new:N \l_@@_skip_empty_lines_bool
637 \bool_set_true:N \l_@@_skip_empty_lines_bool
638 \bool_new:N \l_@@_line_numbers_absolute_bool
639 \tl_new:N \l_@@_line_numbers_format_tl
640 \tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
641 \bool_new:N \l_@@_label_empty_lines_bool
642 \bool_set_true:N \l_@@_label_empty_lines_bool
643 \int_new:N \l_@@_number_lines_start_int
644 \str_new:N \l_@@_line_numbers_position_str
645 \str_set:Nn \l_@@_line_numbers_position_str { left }
646 \bool_new:N \l_@@_resume_bool
647 \bool_new:N \l_@@_split_on_empty_lines_bool
648 \bool_new:N \l_@@_splittable_on_empty_lines_bool
649 \bool_new:N \g_@@_label_as_zlabel_bool

650 \keys_define:nn { PitonOptions / marker }
651 {
652   beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
653   beginning .value_required:n = true ,
654   end .cs_set:Np = \@@_marker_end:n #1 ,
655   end .value_required:n = true ,
656   include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
657   include-lines .default:n = true ,
658   unknown .code:n = \@@_error:n { Unknown~key~for~marker }
659 }

660 \keys_define:nn { PitonOptions / line-numbers }
661 {
662   true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
663   false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
664
665   start .code:n =
666     \bool_set_true:N \l_@@_line_numbers_bool
667     \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
668   start .value_required:n = true ,
669
670   skip-empty-lines .code:n =
671     \bool_if:NF \l_@@_in_PitonOptions_bool
672     { \bool_set_true:N \l_@@_line_numbers_bool }
673     \str_if_eq:nnTF { #1 } { false }
674     { \bool_set_false:N \l_@@_skip_empty_lines_bool }
675     { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
676   skip-empty-lines .default:n = true ,
677
678   label-empty-lines .code:n =
679     \bool_if:NF \l_@@_in_PitonOptions_bool
680     { \bool_set_true:N \l_@@_line_numbers_bool }
681     \str_if_eq:nnTF { #1 } { false }
682     { \bool_set_false:N \l_@@_label_empty_lines_bool }
683     { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
684   label-empty-lines .default:n = true ,
685
686   absolute .code:n =
687     \bool_if:NTF \l_@@_in_PitonOptions_bool
688     { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
689     { \bool_set_true:N \l_@@_line_numbers_bool }
690     \bool_if:NT \l_@@_in_PitonInputFile_bool
691     {
692       \bool_set_true:N \l_@@_line_numbers_absolute_bool
693       \bool_set_false:N \l_@@_skip_empty_lines_bool
```

```

694     } ,
695     absolute .value_forbidden:n = true ,
696
697     resume .code:n =
698         \bool_set_true:N \l_@@_resume_bool
699         \bool_if:NF \l_@@_in_PitonOptions_bool
700         { \bool_set_true:N \l_@@_line_numbers_bool } ,
701     resume .value_forbidden:n = true ,
702
703     sep .dim_set:N = \l_@@_numbers_sep_dim ,
704     sep .value_required:n = true ,
705
706     step .int_set:N = \l_@@_numbers_step_int ,
707     step .value_required:n = true ,
708
709     position .choices:nn = { left , right }
710     { \str_set_eq:NN \l_@@_line_numbers_position_str \l_keys_choice_tl } ,
711     position .value_required:n = true ,
712
713     format .tl_set:N = \l_@@_line_numbers_format_tl ,
714     format .value_required:n = true ,
715
716     unknown .code:n =
717         \@@_unknown_key:nn
718         { PitonOptions / line-numbers }
719         { Unknown~key~for~line-numbers }
720
721 }

```

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

```

722 \keys_define:nn { PitonOptions }
723 {
724     indentations-for-Foxit .choices:nn = { true , false }
725     {
726         \tl_if_eq:VnTF \l_keys_value_tl { true }
727         { \@@_define_leading_space_Foxit: }
728         { \@@_define_leading_space_normal: }
729     } ,
730     box .choices:nn = { c , t , b , m }
731     { \str_set_eq:NN \l_@@_box_str \l_keys_choice_tl } ,
732     box .default:n = c ,
733     break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
734     break-strings-anywhere .default:n = true ,
735     break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,
736     break-numbers-anywhere .default:n = true ,

```

First, we put keys that should be available only in the preamble.

```

737     detected-commands .code:n =
738         \clist_if_in:nnTF { #1 } { rowcolor }
739         {
740             \@@_error:n { rowcolor~in~detected-commands }
741             \clist_set:Nn \l_tmpa_clist { #1 }
742             \clist_remove_all:Nn \l_tmpa_clist { rowcolor }
743             \clist_put_right:No \l_@@_detected_commands_clist \l_tmpa_clist
744         }
745         { \clist_put_right:Nn \l_@@_detected_commands_clist { #1 } } ,
746     detected-commands .value_required:n = true ,
747     detected-commands .usage:n = preamble ,
748     vertical-detected-commands .code:n = \@@_vertical_commands:n { #1 } ,
749     vertical-detected-commands .value_required:n = true ,
750     vertical-detected-commands .usage:n = preamble ,
751     raw-detected-commands .code:n =
752         \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 } ,

```

```

753 raw-detected-commands .value_required:n = true ,
754 raw-detected-commands .usage:n = preamble ,
755 detected-beamer-commands .code:n =
756   \@@_error_if_not_in_beamer:
757   \clist_put_right:Nn \l_@@_beamer_commands_clist { #1 } ,
758 detected-beamer-commands .value_required:n = true ,
759 detected-beamer-commands .usage:n = preamble ,
760 detected-beamer-environments .code:n =
761   \@@_error_if_not_in_beamer:
762   \clist_put_right:Nn \l_@@_beamer_environments_clist { #1 } ,
763 detected-beamer-environments .value_required:n = true ,
764 detected-beamer-environments .usage:n = preamble ,

```

Remark that the command `\lua_escape:n` is fully expandable. That's why we use `\lua_now:e`.

```

765 begin-escape .code:n =
766   \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
767 begin-escape .value_required:n = true ,
768 begin-escape .usage:n = preamble ,
769
770 end-escape .code:n =
771   \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
772 end-escape .value_required:n = true ,
773 end-escape .usage:n = preamble ,
774
775 begin-escape-math .code:n =
776   \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
777 begin-escape-math .value_required:n = true ,
778 begin-escape-math .usage:n = preamble ,
779
780 end-escape-math .code:n =
781   \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
782 end-escape-math .value_required:n = true ,
783 end-escape-math .usage:n = preamble ,
784
785 comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
786 comment-latex .value_required:n = true ,
787 comment-latex .usage:n = preamble ,
788
789 label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
790 label-as-zlabel .default:n = true ,
791 label-as-zlabel .usage:n = preamble ,
792
793 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
794 math-comments .default:n = true ,
795 math-comments .usage:n = preamble ,

```

Now, general keys.

```

796 language .code:n =
797   \str_set:Ne \l_piton_language_str { \str_lowercase:n { #1 } } ,
798 language .value_required:n = true ,
799 path .code:n =
800   \seq_clear:N \l_@@_path_seq
801   \clist_map_inline:nn { #1 }
802   {
803     \str_set:Nn \l_tmpa_str { ##1 }
804     \seq_put_right:No \l_@@_path_seq { \l_tmpa_str }
805   } ,
806 path .value_required:n = true ,

```

The initial value of the key `path` is not empty: it's `.`, that is to say a comma separated list with only one component which is `.`, the current directory.

```

807 path .initial:n = . ,
808 path-write .str_set:N = \l_@@_path_write_str ,
809 path-write .value_required:n = true ,

```

```

810 font-command      .tl_set:N          = \l_@@_font_command_tl ,
811 font-command      .value_required:n = true ,
812 gobble            .int_set:N          = \l_@@_gobble_int ,
813 gobble            .default:n          = -1 ,
814 auto-gobble       .code:n             = \int_set:Nn \l_@@_gobble_int { -1 } ,
815 auto-gobble       .value_forbidden:n = true ,
816 env-gobble        .code:n             = \int_set:Nn \l_@@_gobble_int { -2 } ,
817 env-gobble        .value_forbidden:n = true ,
818 tabs-auto-gobble  .code:n             = \int_set:Nn \l_@@_gobble_int { -3 } ,
819 tabs-auto-gobble  .value_forbidden:n = true ,
820
821 splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
822 splittable-on-empty-lines .default:n  = true ,
823
824 split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,
825 split-on-empty-lines .default:n  = true ,
826
827 split-separation .tl_set:N          = \l_@@_split_separation_tl ,
828 split-separation .value_required:n = true ,
829
830 add-to-split-separation .code:n =
831   \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
832 add-to-split-separation .value_required:n = true ,
833
834 marker .code:n =
835   \bool_lazy_or:nnTF
836     \l_@@_in_PitonInputFile_bool
837     \l_@@_in_PitonOptions_bool
838     { \keys_set:nn { PitonOptions / marker } { #1 } }
839     { \@@_error:n { Invalid~key } } ,
840 marker .value_required:n = true ,
841
842 line-numbers .code:n =
843   \keys_set:nn { PitonOptions / line-numbers } { #1 } ,
844 line-numbers .default:n = true ,
845
846 splittable      .int_set:N          = \l_@@_splittable_int ,
847 splittable      .default:n          = 1 ,
848 background-color .code:n =
849   \clist_set:Nn \l_@@_bg_color_clist { #1 }

```

We keep the lenght of the clist `\l_@@_bg_color_clist` in a counter for efficiency only.

```

850   \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
851 background-color .value_required:n = true ,
852 prompt-background-color .tl_set:N      = \l_@@_prompt_bg_color_tl ,
853 prompt-background-color .value_required:n = true ,

```

With the tuning `write=false`, the content of the environment won't be parsed and won't be printed on the PDF. However, the Lua variables `piton.last_code` and `piton.last_language` will be set (and, hence, `piton.get_last_code` will be operational). The keys `join` and `write` will be honoured.

```

854 print .bool_set:N = \l_@@_print_bool ,
855 print .value_required:n = true ,
856
857 width .code:n =
858   \str_if_eq:nnTF { #1 } { min }
859   {
860     \bool_set_true:N \l_@@_minimize_width_bool
861     \dim_zero:N \l_@@_width_dim
862   }
863   {
864     \bool_set_false:N \l_@@_minimize_width_bool
865     \dim_set:Nn \l_@@_width_dim { #1 }
866   } ,
867 width .value_required:n = true ,
868

```

```

869 max-width .code:n =
870     \bool_set_true:N \l_@@_minimize_width_bool
871     \dim_set:Nn \l_@@_width_dim { #1 } ,
872 max-width .value_required:n = true ,
873
874 paperclip .code:n =
875     \bool_set_true:N \l_@@_paperclip_bool
876     \tl_if_novalue:nTF { #1 }
877         { \str_set:Nn \l_@@_paperclip_str { } }
878         { \str_set:Nn \l_@@_paperclip_str { #1 } } ,
879
880 annotation .bool_set:N = \l_@@_annotation_bool ,
881 annotation .default:n = true ,
882
883 write .str_set:N = \l_@@_write_str ,
884 write .value_required:n = true ,
885 no-write .code:n = \str_set_eq:NN \l_@@_write_str \c_empty_str ,
886 no-write .value_forbidden:n = true ,
887 join .code:n =
888     \str_set:Nn \l_@@_join_str { #1 }
889     \seq_if_in:NnF \g_@@_join_seq { #1 }
890         { \seq_gput_right:No \g_@@_join_seq { #1 } } ,
891 join .value_required:n = true ,
892 join-separation .str_set:N = \l_@@_join_separation_str ,
893 join-separation .value_required:n = true ,
894 no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,
895 no-join .value_forbidden:n = true ,
896
897 left-margin .code:n =
898     \str_if_eq:nnTF { #1 } { auto }
899     {
900         \dim_zero:N \l_@@_left_margin_dim
901         \bool_set_true:N \l_@@_left_margin_auto_bool
902     }
903     {
904         \dim_set:Nn \l_@@_left_margin_dim { #1 }
905         \bool_set_false:N \l_@@_left_margin_auto_bool
906     } ,
907 left-margin .value_required:n = true ,
908
909 right-margin .code:n =
910     \str_if_eq:nnTF { #1 } { auto }
911     {
912         \dim_zero:N \l_@@_right_margin_dim
913         \bool_set_true:N \l_@@_right_margin_auto_bool
914     }
915     {
916         \dim_set:Nn \l_@@_right_margin_dim { #1 }
917         \bool_set_false:N \l_@@_right_margin_auto_bool
918     } ,
919 right-margin .value_required:n = true ,
920
921 tab-size .int_set:N = \l_@@_tab_size_int ,
922 tab-size .value_required:n = true ,
923 show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
924 show-spaces .value_forbidden:n = true ,
925 show-spaces-in-strings .code:n =
926     \tl_set:Nn \l_@@_space_in_string_tl { } , % U+2423
927 show-spaces-in-strings .value_forbidden:n = true ,
928 break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
929 break-lines-in-Piton .default:n = true ,
930 break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
931 break-lines-in-piton .default:n = true ,

```

```

932 break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
933 break-lines .value_forbidden:n = true ,
934 indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,
935 indent-broken-lines .default:n = true ,
936 end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
937 end-of-broken-line .value_required:n = true ,
938 continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
939 continuation-symbol .value_required:n = true ,
940 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
941 continuation-symbol-on-indentation .value_required:n = true ,
942
943 first-line .code:n = \@@_in_PitonInputFile:n
944 { \int_set:Nn \l_@@_first_line_int { #1 } } ,
945 first-line .value_required:n = true ,
946
947 last-line .code:n = \@@_in_PitonInputFile:n
948 { \int_set:Nn \l_@@_last_line_int { #1 } } ,
949 last-line .value_required:n = true ,
950
951 begin-range .code:n = \@@_in_PitonInputFile:n
952 { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
953 begin-range .value_required:n = true ,
954
955 end-range .code:n = \@@_in_PitonInputFile:n
956 { \str_set:Nn \l_@@_end_range_str { #1 } } ,
957 end-range .value_required:n = true ,
958
959 range .code:n = \@@_in_PitonInputFile:n
960 {
961   \str_set:Nn \l_@@_begin_range_str { #1 }
962   \str_set:Nn \l_@@_end_range_str { #1 }
963 } ,
964 range .value_required:n = true ,
965
966 env-used-by-split .code:n =
967   \lua_now:n { piton.env_used_by_split = '#1' } ,
968 env-used-by-split .initial:n = Piton ,
969
970 resume .meta:n = line-numbers/resume ,
971
972 unknown .code:n =
973   \@@_unknown_key:nn
974   { PitonOptions }
975   { Unknown~key~for~PitonOptions } ,
976
977 % deprecated
978 all-line-numbers .code:n =
979   \bool_set_true:N \l_@@_line_numbers_bool
980   \bool_set_false:N \l_@@_skip_empty_lines_bool ,
981 rounded-corners .code:n =
982   \AtBeginDocument
983   {
984     \IfPackageLoadedTF { tikz }
985       { \dim_set:Nn \l_@@_rounded_corners_dim { #1 } }
986       { \@@_err_rounded_corners_without_Tikz: }
987   } ,
988 rounded-corners .default:n = 4 pt
989 }
990 \hook_gput_code:nnn { begindocument } { . }
991 {
992   \IfPackageLoadedTF { tcolorbox }
993   {
994     \pgfkeysifdefined { / tcb / libload / breakable }

```



```

995     {
996         \keys_define:nn { PitonOptions }
997         {
998             tcolorbox .bool_set:N = \l_@@_tcolorbox_bool ,
999             tcolorbox .default:n = true
1000         }
1001     }
1002     {
1003         \keys_define:nn { PitonOptions }
1004         { tcolorbox .code:n = \@@_error:n { library~breakable~not~loaded } }
1005     }
1006 }
1007 {
1008     \keys_define:nn { PitonOptions }
1009     { tcolorbox .code:n = \@@_error:n { tcolorbox~not~loaded } }
1010 }
1011 }

1012 \cs_new_protected:Npn \@@_err_rounded_corners_without_Tikz:
1013 {
1014     \@@_error:n { rounded-corners~without~Tikz }
1015     \cs_gset:Npn \@@_err_rounded_corners_without_Tikz: { }
1016 }

1017 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
1018 {
1019     \bool_if:NTF \l_@@_in_PitonInputFile_bool
1020     { #1 }
1021     { \@@_error:n { Invalid-key } }
1022 }

1023 \NewDocumentCommand \PitonOptions { m }
1024 {
1025     \bool_set_true:N \l_@@_in_PitonOptions_bool
1026     \keys_set:nn { PitonOptions } { #1 }
1027     \bool_set_false:N \l_@@_in_PitonOptions_bool
1028 }

```

When using `\NewPitonEnvironment` a user may use `\PitonOptions` inside. However, the set of keys available should be different that in standard `\PitonOptions`. That's why we define a version of `\PitonOptions` with no restriction on the set of available keys and we will link that version to `\PitonOptions` in such environment.

```

1029 \NewDocumentCommand \@@_fake_PitonOptions { }
1030 { \keys_set:nn { PitonOptions } }

```

## 2.6 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`) whereas the counter `\g_@@_line_int` previously defined is *not* used for that functionality.

```

1031 \int_new:N \g_@@_visual_line_int
1032 \cs_new_protected:Npn \@@_incr_visual_line:
1033 {
1034     \bool_if:NF \l_@@_skip_empty_lines_bool
1035     { \int_gincr:N \g_@@_visual_line_int }
1036 }

```

The following command will be used when the numbers of lines are printed on the left (`line-numbers/position=left`). The number of line is in the counter `\g_@@_visual_line_int`.

```

1037 \cs_new_protected:Npn \@@_print_number_left:
1038 {
1039   \hbox_overlap_left:n
1040   {
1041     \@@_actually_print_number:n { \int_to_arabic:n { \g_@@_visual_line_int } }
1042     \skip_horizontal:N \l_@@_numbers_sep_dim
1043   }
1044 }
```

The following command will be used when the numbers of lines are printed on the right (`line-numbers/position=right`). The number of line is in `\l_tmpa_tl`.

```

1045 \cs_new_protected:Npn \@@_print_number_right:
1046 {
1047   \hbox_overlap_left:n
1048   {
1049     \@@_actually_print_number:n { \l_tmpa_tl }
1050     \int_compare:nNnT \l_@@_bg_colors_int > 0
1051     { \skip_horizontal:n { 0.1 em } }
1052   }
1053 }
```

`\@@_actually_print_number:` itself prints the number without the `\hbox_overlap_left:n`. It is used by both `\@@_print_number_left:` and `\@@_print_number_right:`

```

1054 \cs_new_protected:Npn \@@_actually_print_number:n #1
1055 {
1056   \group_begin:
1057   \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
```

We put braces. Thus, the user may use the key `line-numbers/format` with a value such as `\fbox`.

```

1058   \l_@@_line_numbers_format_tl { #1 }
1059   \pdfextension literal { EMC }
1060   \group_end:
1061 }
```

## 2.7 The main commands and environments for the end user

```

1062 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
1063 {
1064   \tl_if_novalue:nTF { #3 }
```

The last argument is provided by curryfication.

```

1065   { \@@_NewPitonLanguage:nnn { #1 } { #2 } }
```

The two last arguments are provided by curryfication.

```

1066   { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
1067 }
```

The following property list will contain the definitions of the computer languages as provided by the end user. However, if a language is defined over another base language, the corresponding list will contain the *whole* definition of the language.

```

1068 \prop_new:N \g_@@_languages_prop
```

```

1069 \keys_define:nn { NewPitonLanguage }
1070 {
1071   morekeywords .code:n = ,
1072   otherkeywords .code:n = ,
1073   sensitive .code:n = ,
1074   keywordsprefix .code:n = ,
1075   moretexcs .code:n = ,
```

```

1076   morestring .code:n = ,
1077   morecomment .code:n = ,
1078   moredelim .code:n = ,
1079   moredirectives .code:n = ,
1080   tag .code:n = ,
1081   alsodigit .code:n = ,
1082   alsoletter .code:n = ,
1083   alsoother .code:n = ,
1084   unknown .code:n = \@@_error:n { Unknown~key~NewPitonLanguage }
1085 }

```

The function `\@@_NewPitonLanguage:nnn` will be used when the language is *not* defined above a base language (and a base dialect).

```

1086 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
1087 {

```

We store in `\l_tmpa_tl` the name of the language with the potential dialect, that is to say, for example : `[AspectJ]{Java}`. We use `\tl_if_blank:nF` because the end user may have written `\NewPitonLanguage[ ]{Java}{...}`.

```

1088   \tl_set:Nx \l_tmpa_tl
1089   {
1090     \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
1091     \str_lowercase:n { #2 }
1092   }

```

The following set of keys is only used to raise an error when a key is unknown!

```

1093   \keys_set:nn { NewPitonLanguage } { #3 }

```

We store in LaTeX the definition of the language because some languages may be defined with that language as base language.

```

1094   \prop_gput:Non \g_@@_languages_prop \l_tmpa_tl { #3 }

```

The Lua part of the package `piton` will be loaded in a `\AtBeginDocument`. Hence, we will put also in a `\AtBeginDocument` the use of the Lua function `piton.new_language` (which does the main job).

```

1095   \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
1096 }
1097 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
1098 {
1099   \hook_gput_code:nnn { begindocument } { . }
1100   { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }
1101 }
1102 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }

```

Now the case when the language is defined upon a base language.

```

1103 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnnn #1 #2 #3 #4 #5
1104 {

```

We store in `\l_tmpa_tl` the name of the base language with the dialect, that is to say, for example : `[AspectJ]{Java}`. We use `\tl_if_blank:nF` because the end user may have used `\NewPitonLanguage[Handel]{C}[ ]{C}{...}`

```

1105   \tl_set:Nx \l_tmpa_tl
1106   {
1107     \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
1108     \str_lowercase:n { #4 }
1109   }

```

We retrieve in `\l_tmpb_tl` the definition (as provided by the end user) of that base language. Caution: `\g_@@_languages_prop` does not contain all the languages provided by `piton` but only those defined by using `\NewPitonLanguage`.

```

1110   \prop_get:NoNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl

```

We can now define the new language by using the previous function.

```

1111   { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
1112   { \@@_error:n { Language~not~defined } }
1113 }

```

```
1114 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4
```

In the following line, we write #4,#3 and not #3,#4 because we want that the keys which correspond to base language appear before the keys which are added in the language we define.

```
1115 { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
1116 \cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }
```

```
1117 \NewDocumentCommand { \piton } { }
1118 { \peek_meaning:NTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }
1119 \NewDocumentCommand { \@@_piton_standard } { m }
1120 {
1121   \group_begin:
1122   \tl_if_eq:NnF \l_@@_space_in_string_tl { } {
1123     {
```

Remind that, when `break-strings-anywhere` is in force, multiple commands `\-` will be inserted between the characters of the string to allow the breaks. The `\exp_not:N` before `\space` is mandatory.

```
1124     \bool_lazy_or:nnT
1125     \l_@@_break_lines_in_piton_bool
1126     \l_@@_break_strings_anywhere_bool
1127     { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
1128   }
```

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

```
1129 \automatichyphenmode = 1
```

Remark that the argument of `\piton` (with the normal syntax) is expanded in the TeX sens, (see the `\tl_set:Ne` below) and that's why we can provide the following escapes to the end user:

```
1130 \cs_set_eq:NN \ \ \c_backslash_str
1131 \cs_set_eq:NN \% \c_percent_str
1132 \cs_set_eq:NN \{ \c_left_brace_str
1133 \cs_set_eq:NN \} \c_right_brace_str
1134 \cs_set_eq:NN \$ \c_dollar_str
```

The standard command `\_` is *not* expandable and we need here expandable commands. With the following code, we define an expandable command.

```
1135 \cs_set_eq:cN { ~ } \space
1136 \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
```

We redefine `\rowcolor` inside of `\piton` commands to do nothing.

```
1137 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1138 \tl_set:Ne \l_tmpa_tl
1139 {
1140   \lua_now:e
1141   { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1142   { #1 }
1143 }
1144 \bool_if:NTF \l_@@_show_spaces_bool
1145 { \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { } } % U+2423
1146 {
1147   \bool_if:NT \l_@@_break_lines_in_piton_bool
```

With the following line, the spaces of catacode 12 (which were not breakable) are replaced by `\space`, and, thus, become breakable.

```
1148   { \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl \space }
1149 }
```

The command `\text` is provided by the package `amstext` (loaded by `piton`).

```
1150 \if_mode_math:
1151   \text { \l_@@_font_command_tl \l_tmpa_tl }
1152 \else:
1153   \l_@@_font_command_tl \l_tmpa_tl
1154 \fi:
1155 \group_end:
```

```

1156 }

1157 \NewDocumentCommand { \@@_piton_verbatim } { v }
1158 {
1159   \group_begin:
1160   \automatichyphenmode = 1
1161   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

We redefine \rowcolor inside of \piton commands to do nothing.

```

1162   \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1163   \tl_set:Ne \l_tmpa_tl
1164   {
1165     \lua_now:e
1166     { piton.Parse('\l_piton_language_str',token.scan_string()) }
1167     { #1 }
1168   }
1169   \bool_if:NT \l_@@_show_spaces_bool
1170   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1171   \if_mode_math:
1172     \text { \l_@@_font_command_tl \l_tmpa_tl }
1173   \else:
1174     \l_@@_font_command_tl \l_tmpa_tl
1175   \fi:
1176   \group_end:
1177 }

```

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

```

1178 \cs_new_protected:Npn \@@_piton:n #1
1179 { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }
1180
1181 \cs_new_protected:Npn \@@_piton_i:n #1
1182 {
1183   \group_begin:
1184   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
1185   \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1186   \cs_set:cpn { pitonStyle _ Prompt } { }
1187   \cs_set_eq:NN \@@_leading_space: \space
1188   \cs_set_eq:NN \@@_trailing_space: \space
1189   \tl_set:Ne \l_tmpa_tl
1190   {
1191     \lua_now:e
1192     { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1193     { #1 }
1194   }
1195   \bool_if:NT \l_@@_show_spaces_bool
1196   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1197   \@@_replace_spaces:o \l_tmpa_tl
1198   \group_end:
1199 }

```

`\@@_pre_composition:` will be used both in `\PitonInputFile` and in the environments such as `{Piton}`.

```

1200 \cs_new_protected:Npn \@@_pre_composition:
1201 {
1202   \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
1203   {
1204     \dim_set_eq:NN \l_@@_width_dim \linewidth

```

When the key `box` is used, `width=min` is activated (except when `width` has been used with a numerical value).

```

1205     \str_if_empty:NF \l_@@_box_str
1206     { \bool_set_true:N \l_@@_minimize_width_bool }
1207 }

```

We compute `\l_@@_listing_width_dim`. However, if `max-width` is used (or `width=min` which uses `max-width`), that length will be computed again in `\@@_create_output_box`: but **even in the case**, we have to compute that value now (because the maximal width set by `max-width` may be reached by some lines of the listing—and those lines would be wrapped).

```

1208     \dim_set:Nn \l_@@_listing_width_dim
1209     {
1210         \bool_if:NTF \l_@@_tcolorbox_bool
1211         {
1212             \l_@@_width_dim -
1213             ( \kvtcb@left@rule
1214             + \kvtcb@leftupper
1215             + \kvtcb@boxsep * 2
1216             + \kvtcb@rightupper
1217             + \kvtcb@right@rule )
1218         }
1219         { \l_@@_width_dim }
1220     }

1221     \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1222     \automatichyphenmode = 1
1223     \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1224     \g_@@_def_vertical_commands_tl
1225     \int_gzero:N \g_@@_line_int
1226     \int_gzero:N \g_@@_nb_lines_int
1227     \dim_zero:N \parindent
1228     \dim_zero:N \lineskip
1229     \dim_zero:N \parskip

1230
1231     % added 2026-01-02
1232     \seq_gclear:N \g_@@_visual_line_numbers_seq
1233
1234     \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

For efficiency, we keep in `\l_@@_bg_colors_int` the length of `\l_@@_bg_color_clist`.

```

1235     \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1236     { \bool_set_true:N \l_@@_bg_bool }
1237     \bool_gset_false:N \g_@@_rowcolor_inside_bool
1238     \IfPackageLoadedTF { zref-base }
1239     {
1240         \bool_if:NTF \g_@@_label_as_zlabel_bool
1241         { \cs_set_eq:NN \label \@@_zlabel:n }
1242         { \cs_set_eq:NN \label \@@_label:n }
1243         \cs_set_eq:NN \zlabel \@@_zlabel:n
1244     }
1245     { \cs_set_eq:NN \label \@@_label:n }
1246     \l_@@_font_command_tl
1247 }

```

When the parameters `line-numbers`, `line-numbers/position=left` and `left-margin` are in force (or if `line-numbers`, `line-numbers=right` and `right-margin` are in force), 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` (or `right-margin`).

The command `\@@_compute_margin:N` will do that job.

It's argument must be either `\l_@@_left_margin_dim` either `\l_@@_right_margin_dim`.

```

1248     \cs_new_protected:Npn \@@_compute_margin:N #1
1249     {
1250         \use:e
1251         {

```

```

1252     \bool_if:NTF \l_@@_skip_empty_lines_bool
1253     { \lua_now:n { piton.CountNonEmptyLines(token.scan_argument()) } }
1254     { \lua_now:n { piton.CountLines(token.scan_argument()) } }
1255     { \l_@@_listing_tl }
1256   }
1257   \hbox_set:Nn \l_tmpa_box
1258   {
1259     \l_@@_line_numbers_format_tl
1260     \int_to_arabic:n
1261     {
1262       \g_@@_visual_line_int
1263       +
1264       \bool_if:NTF \l_@@_skip_empty_lines_bool
1265       { \l_@@_nb_non_empty_lines_int }
1266       { \g_@@_nb_lines_int }
1267     }
1268   }
1269   \dim_set:Nn #1 { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1270 }

```

The following command computes `\l_@@_code_width_dim`.

It will be used only once (in `\@@_create_output_box:`).

If there is a background (even a background with the color `none`), we subtract 0.5 em on both sides. However, if there is a left margin or a right margin, we use those margins. If the key `left-margin` has been used with the special value `auto` (this is meaningful only in conjunction with the key `line-numbers` and a value of `line-numbers/position` equal to `left`), the actual value for the left margin has yet computed (and stored in `left-margin`). Idem for the right margin.

```

1271 \cs_new_protected:Npn \@@_compute_code_width:
1272 {
1273   \dim_set:Nn \l_@@_code_width_dim
1274   {
1275     \l_@@_listing_width_dim
1276     -
1277     (
1278       \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1279       {
1280         \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1281         { \l_@@_left_margin_dim }
1282         { 0.5 em }
1283       }
1284       +
1285       \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1286       { \l_@@_right_margin_dim }
1287       { 0.5 em }
1288     )
1289     { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1290   }
1291 }

```

The following command computes `\l_@@_listing_width_dim` and it will be used when `max-width` (or `width=min`) is used. Remind that the key `box` sets `width=min` (except when `width` is used with a numerical value).

It will be used only once (in `\@@_create_output_box:`).

The computation is the inverse of the computation done in `\@@_compute_code_width:`.

```

1292 \cs_new_protected:Npn \@@_recompute_listing_width:
1293 {
1294   \dim_set:Nn \l_@@_listing_width_dim
1295   {
1296     \box_wd:N \g_@@_output_box
1297     +
1298     \int_compare:nNnTF \l_@@_bg_colors_int > \c_zero_int
1299     {

```

```

1300         \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1301         { \l_@@_left_margin_dim }
1302         { 0.5 em }
1303     +
1304     \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1305     { \l_@@_right_margin_dim }
1306     { 0.5 em }
1307 }
1308 { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1309 }
1310 }

```

```

1311 \cs_new_protected:Npn \@@_store_body:n #1
1312 {

```

Now, we have to replace all the occurrences of `\obeyedline` by a character of end of line (`\r` in the strings of Lua).

```

1313     \tl_set:Nc \obeyedline { \char_generate:nn { 13 } { 11 } }
1314     \tl_set:Nc \l_@@_listing_tl { #1 }
1315     \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1316 }

```

The first argument of the following macro is one of the four strings: New, Renew, Provide and Declare.

```

1317 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nnnnn
1318 {
1319     \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1320     {
1321         \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1322         #4
1323         \@@_pre_composition:
1324         \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1325         {
1326             \int_gset:Nn \g_@@_visual_line_int
1327             { \l_@@_number_lines_start_int - 1 }
1328         }
1329         \bool_if:NT \g_@@_beamer_bool
1330         { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1331         \bool_if:NT \g_@@_footnote_bool \savenotes
1332         \@@_composition:
1333         \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1334         { \@@_create_paperclip_annotation: }
1335         \bool_if:NT \g_@@_footnote_bool \endsavenotes
1336         #5
1337     }
1338     { \ignorespacesafterend }
1339 }

```

`\marginalia` is a command of the package `marginalia` (loaded by `piton`).

```

1340 \cs_new_protected:Npn \@@_create_paperclip_annotation:
1341 {
1342     \marginalia
1343     {
1344         \vspace* { - 0.8 em }
1345         \hbox:n
1346         {
1347             \vrule~height~0~pt~depth~12~pt~width~0~pt
1348             \bool_if:NT \l_@@_annotation_bool
1349             {
1350                 \lua_now:n
1351                 {

```

The function `piton.utf16` does a conversion from utf8 to utf16 big endian encoded in hexadecimal (with the BOM of big endian), which is suitable to be put in a string between angular brackets of the PDF. It's easier for a stream!



```

1352         pdf.immediateobj
1353         ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1354     }
1355     \pdfextension annot~width~5pt~height~10pt~depth~0pt
1356     {
1357         /Subtype /Text
1358         /Contents~\pdf_object_ref_last:
1359         /Name /Note
1360         /Subj (Computer~listing)

```

The following tries to specify that the note should not receive answers (since it is meant for an easy copy-past of the computer listing).

```

1361         /ReplyType /Group

```

Adds the bit 10 which means LockedContents.

```

1362         /F~512
1363         /C [0.8~0.8~0.8]
1364     }
1365     \hspace* { 7 mm }
1366 }
1367 \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1368 }
1369 }
1370 }

```

```

1371 \cs_new_protected:Npn \@@_create_paperclip:
1372 {
1373     \str_if_empty:NT \l_@@_paperclip_str
1374     {
1375         \int_gincr:N \g_@@_paperclip_int
1376         \str_set:Ne \l_@@_paperclip_str { listing~\int_use:N \g_@@_paperclip_int .txt }
1377     }

```

Here, we don't understand why the tostring is mandatory.

```

1378     \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1379     \box_move_down:nn
1380     { 10 pt }
1381     {
1382         \hbox:n
1383         {
1384             \pdfextension annot~width~10pt~height~20pt~depth~0pt
1385             {
1386                 /Subtype /FileAttachment
1387                 /Name /Paperclip
1388                 /F~8 % no zoom

```

/Contents will be used as info-bulle and description of the file in the panel of the embedded files.

```

1389         /Contents (The~computer~listing)
1390         /FS <<
1391             /Type /Filespec
1392             /F (\l_@@_paperclip_str)
1393             /EF << /F~\pdf_object_ref_last: >>
1394             /AFRelationship /Supplement
1395             >>
1396         }
1397     }
1398 }
1399 }

```

For the following commands, the arguments are provided by curryfication.

```

1400 \NewDocumentCommand { \NewPitonEnvironment } { }
1401 { \@@_DefinePitonEnvironment:nnnnn { New } }
1402 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1403 { \@@_DefinePitonEnvironment:nnnnn { Declare } }

```

```

1404 \NewDocumentCommand { \RenewPitonEnvironment } { }
1405 { \@@_DefinePitonEnvironment:nnnnn { Renew } }

1406 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1407 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

1408 \cs_new_protected:Npn \@@_translate_beamer_env:n
1409 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1410 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1411 \cs_new_protected:Npn \@@_composition:
1412 {
1413   \str_if_empty:NT \l_@@_box_str
1414   {
1415     \mode_if_vertical:F
1416     { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1417   }

1418   \bool_if:NT \l_@@_line_numbers_bool
1419   {
1420     \bool_lazy_and:nnT
1421     { \l_@@_left_margin_auto_bool }
1422     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { left } }
1423     { \@@_compute_margin:N \l_@@_left_margin_dim }
1424     \bool_lazy_and:nnT
1425     { \l_@@_right_margin_auto_bool }
1426     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { right } }
1427     { \@@_compute_margin:N \l_@@_right_margin_dim }
1428   }

1429   \lua_now:e
1430   {
1431     piton.join_separation = "\l_@@_join_separation_str"
1432     piton.join = "\l_@@_join_str"
1433     piton.write = "\l_@@_write_str"
1434     piton.path_write = "\l_@@_path_write_str"
1435   }

1436   \noindent
1437   \bool_if:NTF \l_@@_print_bool
1438   {

```

When `split-on-empty-lines` is in force, each chunk will be formatted by an environment `{Piton}` (or the environment specified by `env-used-by-split`). Within each of these environments, we will come back here (but, of course, `split-on-empty-line` will have been set to `false`). The mechanism “retrieve” is mandatory.

```

1439   \bool_if:NTF \l_@@_split_on_empty_lines_bool
1440   { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1441   {
1442     \@@_create_output_box:

```

Now, the listing has been composed in `\g_@@_output_box` and `\l_@@_listing_width_dim` contains the width of the listing (with the potential margin for the numbers of lines).

```

1443   \bool_if:NTF \l_@@_tcolorbox_bool
1444   {
1445     \str_if_empty:NTF \l_@@_box_str
1446     { \@@_composition_iii: }
1447     { \@@_composition_iv: }
1448   }
1449   {
1450     \str_if_empty:NTF \l_@@_box_str
1451     { \@@_composition_i: }
1452     { \@@_composition_ii: }
1453   }
1454 }
1455 }

```

```

1456     { \@@gobble_parse_no_print:o \l_@@_listing_tl }
1457 }

```

`\@@_composition_i:` is for the main case: the key `tcolorbox` is not used, nor the key `box`.

We can't do a mere `\vbox_unpack:N \g_@@_output_box` because that would not work inside a list of LaTeX (`{\itemize}` or `{\enumerate}`).

The composition in the box `\g_@@_output_box` was mandatory to be able to deal with the case of a conjunction of the keys `width=min` and `background-color=...`.

```

1458 \cs_new_protected:Npn \@@_composition_i:
1459 {

```

First, we “reverse” the box `\g_@@_output_box`: we put in the box `\g_tmpa_box` the boxes present in `\g_@@_output_box`, but in reversed order. The vertical spaces and the penalties are discarded.

```

1460   \box_clear:N \g_tmpa_box

```

The box `\g_@@_line_box` will be used as an auxiliary box.

```

1461   \box_clear_new:N \g_@@_line_box

```

We unpack `\g_@@_output_box` in `\l_tmpa_box` used as a scratched box.

```

1462   \vbox_set:Nn \l_tmpa_box
1463   {
1464     \vbox_unpack_drop:N \g_@@_output_box
1465     \bool_gset_false:N \g_tmpa_bool
1466     \unskip \unskip
1467     \bool_gset_false:N \g_tmpa_bool
1468     \bool_do_until:nn \g_tmpa_bool
1469     {
1470       \unskip \unskip \unskip
1471       \unpenalty \unkern
1472       \box_set_to_last:N \l_@@_line_box
1473       \box_if_empty:NTF \l_@@_line_box
1474       { \bool_gset_true:N \g_tmpa_bool }
1475       {
1476         \vbox_gset:Nn \g_tmpa_box
1477         {
1478           \vbox_unpack:N \g_tmpa_box
1479           \box_use:N \l_@@_line_box
1480         }
1481       }
1482     }
1483   }

```

Now, we will loop over the boxes in `\g_tmpa_box` and compose the boxes in the TeX flow.

```

1484   \bool_gset_false:N \g_tmpa_bool
1485   \int_zero:N \g_@@_line_int
1486   \bool_do_until:nn \g_tmpa_bool
1487   {

```

We retrieve the last box of `\g_tmpa_box` (and store it in `\g_@@_line_box`) and keep the other boxes in `\g_tmpa_box`.

```

1488     \vbox_gset:Nn \g_tmpa_box
1489     {
1490       \vbox_unpack_drop:N \g_tmpa_box
1491       \box_gset_to_last:N \g_@@_line_box
1492     }

```

If the box that we have retrieved is void, that means that, in fact, there is no longer boxes in `\g_tmpa_box` and we will exit the loop.

```

1493     \box_if_empty:NTF \g_@@_line_box
1494     { \bool_gset_true:N \g_tmpa_bool }
1495     {
1496       \box_use:N \g_@@_line_box
1497       \int_gincr:N \g_@@_line_int
1498       \par
1499       \kern -2.5 pt

```

We will determine the penalty by reading the Lua table `piton.lines_status`. That will use the current value of `\g_@@_line_int`.

```
1500 \@@_add_penalty_for_the_line:
```

We now add the instructions corresponding to the *vertical detected commands* that are potentially used in the corresponding line of the listing.

```
1501 \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 }
1502 { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 } }
1503 \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1504 { \mode_leave_vertical: }
1505 }
1506 }
1507 \skip_vertical:n { 2.5 pt }
1508 }
```

`\@@_composition_ii`: will be used when the key `box` is in force but *not* the key `tcolorbox`.

```
1509 \cs_new_protected:Npn \@@_composition_ii:
1510 {
1511 \use:e { \begin { minipage } [ \l_@@_box_str ] }
1512 { \l_@@_listing_width_dim }
```

Here, `\vbox_unpack:N`, instead of `\box_use:N` is mandatory for the vertical position of the box.

```
1513 \vbox_unpack:N \g_@@_output_box
```

`\kern` is mandatory here (`\skip_vertical:n` won't work).

```
1514 \kern 2.5 pt
1515 \end { minipage }
1516 }
```

`\@@_composition_iii`: will be used when the key `tcolorbox` is in force but *not* the key `box`.

```
1517 \cs_new_protected:Npn \@@_composition_iii:
1518 {
1519 \use:e
1520 {
1521 \begin { tcolorbox }
```

Even though we use the key `breakable` of `{tcolorbox}`, our environment will be breakable only when the key `splittable` of `piton` is used.

```
1522 [ breakable , text~width = \l_@@_listing_width_dim ]
1523 }
1524 \par
1525 \vbox_unpack:N \g_@@_output_box
1526 \end { tcolorbox }
1527 }
```

`\@@_composition_iv`: will be used when both keys `tcolorbox` and `box` are in force.

```
1528 \cs_new_protected:Npn \@@_composition_iv:
1529 {
1530 \use:e
1531 {
1532 \begin { tcolorbox }
1533 [
1534 hbox ,
1535 text~width = \l_@@_listing_width_dim ,
1536 nobeforeafter ,
1537 box~align =
1538 \str_case:Nn \l_@@_box_str
1539 {
1540 t { top }
1541 b { bottom }
1542 c { center }
1543 m { center }
1544 }
1545 ]
1546 }
```

```

1547 \box_use:N \g_@@_output_box
1548 \end { tcolorbox }
1549 }

```

The following function will add the correct vertical penalty after a line of code in order to control the breaks of the pages. We use the Lua table `piton.lines_status` which has been written by `piton.ComputeLinesStatus` for this aim. Each line has a “status” (equal to 0, 1 or 2) and that status directly says whether a break is allowed.

```

1550 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1551 {
1552   \int_case:nn
1553   {
1554     \lua_now:e
1555     {
1556       tex.sprint
1557       ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1558     }
1559   }
1560   { 1 { \penalty 100 } 2 \nobreak }
1561 }

```

`\@@_create_output_box:` is used only once, in `\@@_composition:`.

It creates (and modifies when there are backgrounds or numbers of the lines on the right) `\g_@@_output_box`.

```

1562 \cs_new_protected:Npn \@@_create_output_box:
1563 {
1564   \@@_compute_code_width:
1565   \vbox_gset:Nn \g_@@_output_box
1566   { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1567   \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1568   \bool_lazy_any:nT
1569   {
1570     { \int_compare_p:nNn \l_@@_bg_colors_int > { \c_zero_int } }
1571     { \g_@@_rowcolor_inside_bool }
1572     {
1573       \l_@@_line_numbers_bool
1574       &&
1575       \str_if_eq_p:ee { \l_@@_line_numbers_position_str } { right }
1576     }
1577   }
1578   { \@@_add_bg_and_right_nb_to_output_box: }
1579 }

```

We add the backgrounds after the composition of the box `\g_@@_output_box` by a loop over the lines in that box. Idem when the key `line-numbers` is used in conjunction with `line-numbers/position=right`.

The backgrounds will have a width equal to `\l_@@_listing_width_dim`.

That command will be used only once, in `\@@_create_output_box:`.

```

1580 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_output_box:
1581 {
1582   \int_gset_eq:NN \g_@@_line_int \g_@@_nb_lines_int

```

`\l_tmpa_box` is only used to *unpack* the vertical box `\g_@@_output_box`.

```

1583 \vbox_set:Nn \l_tmpa_box
1584 {
1585   \vbox_unpack_drop:N \g_@@_output_box

```

We will raise `\g_tmpa_bool` to exit the loop `\bool_do_until:nn` below.

```

1586   \bool_gset_false:N \g_tmpa_bool
1587   \unskip \unskip

```

We begin the loop.

```

1588   \bool_do_until:nn \g_tmpa_bool

```

```

1589     {
1590         \unskip \unskip \unskip
1591         \int_set_eq:NN \l_tmpa_int \lastpenalty
1592         \unpenalty \unkern

```

In standard TeX (not LuaTeX), the only way to loop over the sub-boxes of a given box is to use the TeX primitive `\lastbox` (via `\box_set_to_last:N` of L3). Of course, it would be interesting to replace that programming by a programming in Lua of LuaTeX...

```

1593         \box_set_to_last:N \l_@@_line_box
1594         \box_if_empty:NTF \l_@@_line_box
1595         { \bool_gset_true:N \g_tmpa_bool }
1596         {

```

`\g_@@_line_int` will be used in `\@@_add_bg_and_right_nb_to_line_and_use:`.

```

1597         \vbox_gset:Nn \g_@@_output_box
1598         {

```

The command `\@@_add_bg_and_right_nb_to_line_and_use:` will add a background to the line (in `\l_@@_line_box`) but will also put the line in the current box. The background will have a width equal to `\l_@@_listing_width_dim`.

```

1599             \@@_add_bg_and_right_nb_to_line_and_use:
1600             \kern -2.5 pt
1601             \penalty \l_tmpa_int
1602             \vbox_unpack:N \g_@@_output_box
1603         }
1604     }
1605     \int_gdecr:N \g_@@_line_int
1606 }
1607 }
1608 }

```

The following will be used when the end user has used `print=false`.

```

1609 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1610 {
1611     \lua_now:e
1612     {
1613         piton.GobbleParseNoPrint
1614         (
1615             '\l_piton_language_str' ,
1616             \int_use:N \l_@@_gobble_int ,
1617             token.scan_argument ( )
1618         )
1619     }
1620 }
1621 \cs_generate_variant:Nn \@@_gobble_parse_no_print:n { o }

```

The following function will be used when the key `split-on-empty-lines` is not in force. It will retrieve the first empty line, gobble the spaces at the beginning of the lines and parse the code. The argument is provided by curryfication.

```

1622 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1623 {
1624     \lua_now:e
1625     {
1626         piton.RetrieveGobbleParse
1627         (
1628             '\l_piton_language_str' ,
1629             \int_use:N \l_@@_gobble_int ,
1630             \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1631             { \int_eval:n { - \l_@@_splittable_int } }
1632             { \int_use:N \l_@@_splittable_int } ,
1633             token.scan_argument ( )
1634         )
1635     }
1636 }

```

```
1637 \cs_generate_variant:Nn \@@_retrieve_gobble_parse:n { o }
```

The following function will be used when the key `split-on-empty-lines` is in force. It will gobble the spaces at the beginning of the lines (if the key `gobble` is in force), then split the code at the empty lines and, eventually, parse the code. The argument is provided by curryfication.

```
1638 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1639 {
1640   \lua_now:e
1641   {
1642     piton.RetrieveGobbleSplitParse
1643     (
1644       '\l_piton_language_str' ,
1645       \int_use:N \l_@@_gobble_int ,
1646       \int_use:N \l_@@_splittable_int ,
1647       token.scan_argument ( )
1648     )
1649   }
1650 }
1651 \cs_generate_variant:Nn \@@_retrieve_gobble_split_parse:n { o }
```

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

```
1652 \bool_if:NTF \g_@@_beamer_bool
1653 {
1654   \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1655   {
1656     \keys_set:nn { PitonOptions } { #2 }
1657     \begin { actionenv } < #1 >
1658   }
1659   { \end { actionenv } }
1660 }
1661 {
1662   \NewPitonEnvironment { Piton } { 0 { } }
1663   { \keys_set:nn { PitonOptions } { #1 } }
1664   { }
1665 }

1666 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1667 {
1668   \mode_if_vertical:F { \par }
1669   \group_begin:
1670   \seq_concat:NNN
1671     \l_file_search_path_seq
1672     \l_@@_path_seq
1673     \l_file_search_path_seq
1674   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1675   {
1676     \@@_input_file:nn { #1 } { #2 }
1677     #4
1678   }
1679   { #5 }
1680   \group_end:
1681 }

1682 \cs_new_protected:Npn \@@_unknown_file:n #1
1683 { \msg_error:nnn { piton } { Unknown~file } { #1 } }
1684 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1685 {
1686   \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1687 }
```

The following line is for `latexmk` (suggestion of Y. Salmon).

```

1688     \iow_log:n { No~file~#3 }
1689     \@@_unknown_file:n { #3 }
1690   }
1691 }
1692 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1693 {
1694   \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1695   {

```

The following line is for latexmk (suggestion of Y. Salmon).

```

1696     \iow_log:n { No~file~#3 }
1697     \@@_unknown_file:n { #3 }
1698   }
1699 }
1700 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1701 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

The following command uses as implicit argument the name of the file in `\l_@@_file_name_str`.

```

1702 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1703 {

```

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

```

1704   \tl_if_novalue:nF { #1 }
1705   {
1706     \bool_if:NTF \g_@@_beamer_bool
1707     { \begin { uncoverenv } < #1 > }
1708     { \@@_error_or_warning:n { overlay~without~beamer } }
1709   }
1710   \group_begin:

```

The following line is to allow tools such as latexmk to be aware that the file read by `\PitonInputFile` is loaded during the compilation of the LaTeX document.

```

1711   \iow_log:e { (\l_@@_file_name_str) }
1712   \int_zero_new:N \l_@@_first_line_int
1713   \int_zero_new:N \l_@@_last_line_int
1714   \int_set_eq:NN \l_@@_last_line_int \c_max_int
1715   \bool_set_true:N \l_@@_in_PitonInputFile_bool
1716   \keys_set:nn { PitonOptions } { #2 }
1717   \bool_if:NT \l_@@_line_numbers_absolute_bool
1718   { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1719   \bool_if:NTF
1720   {
1721     (
1722       \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1723       || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1724     )
1725     && ! \str_if_empty_p:N \l_@@_begin_range_str
1726   }
1727   {
1728     \@@_error_or_warning:n { bad-range-specification }
1729     \int_zero:N \l_@@_first_line_int
1730     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1731   }
1732   {
1733     \str_if_empty:NF \l_@@_begin_range_str
1734     {
1735       \@@_compute_range:
1736       \bool_lazy_or:nnT
1737       \l_@@_marker_include_lines_bool
1738       { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1739       {
1740         \int_decr:N \l_@@_first_line_int
1741         \int_incr:N \l_@@_last_line_int
1742       }
1743     }
1744   }

```



```

1744     }
1745     \@@_pre_composition:
1746     \bool_if:NT \l_@@_line_numbers_absolute_bool
1747     { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1748     \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1749     {
1750         \int_gset:Nn \g_@@_visual_line_int
1751         { \l_@@_number_lines_start_int - 1 }
1752     }

```

The following case arises when the code `line-numbers/absolute` is in force without the use of a marked range.

```

1753     \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1754     { \int_gzero:N \g_@@_visual_line_int }
1755     \lua_now:e
1756     {

```

The following command will store the content of the file (or only a part of that file) in `\l_@@_listing_tl`.

```

1757         piton.ReadFile(
1758             '\l_@@_file_name_str' ,
1759             \int_use:N \l_@@_first_line_int ,
1760             \int_use:N \l_@@_last_line_int )
1761     }
1762     \@@_composition:
1763     \group_end:

```

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

```

1764     \tl_if_novalue:nF { #1 }
1765     { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1766 }

```

The following command computes the values of `\l_@@_first_line_int` and `\l_@@_last_line_int` when `\PitonInputFile` is used with textual markers.

```

1767 \cs_new_protected:Npn \@@_compute_range:
1768 {

```

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

```

1769     \str_set:Ne \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1770     \str_set:Ne \l_tmpb_str { \@@_marker_end:n { \l_@@_end_range_str } }

```

We replace the sequences `\#` which may be present in the prefixes and suffixes added to the markers by the functions `\@@_marker_beginning:n` and `\@@_marker_end:n`.

```

1771     \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1772     \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1773     \lua_now:e
1774     {
1775         piton.ComputeRange
1776         ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1777     }
1778 }

```

## 2.8 The styles

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

```

1779 \NewDocumentCommand { \PitonStyle } { m }
1780 {
1781     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1782     { \use:c { pitonStyle _ #1 } }
1783 }

```

The following variant will be rarely used. It applies only a local style and only when that style exists (no error will be raised when the style does not exist). That command will be used in particular for the language “expl”.

```

1784 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1785   { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1786 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1787   {
1788     \str_clear_new:N \l_@@_SetPitonStyle_option_str
1789     \str_set:Nx \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1790     \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1791       { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1792     \keys_set:nn { piton / Styles } { #2 }
1793   }

1794 \cs_new_protected:Npn \@@_math_scantokens:n #1
1795   { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

1796 \clist_new:N \g_@@_styles_clist
1797 \clist_gset:Nn \g_@@_styles_clist
1798   {
1799     Comment ,
1800     Comment.Internal ,
1801     Comment.LaTeX ,
1802     Discard ,
1803     Exception ,
1804     FormattingType ,
1805     Identifier.Internal ,
1806     Identifier ,
1807     InitialValues ,
1808     Interpol.Inside ,
1809     Keyword ,
1810     Keyword.Governing ,
1811     Keyword.Constant ,
1812     Keyword2 ,
1813     Keyword3 ,
1814     Keyword4 ,
1815     Keyword5 ,
1816     Keyword6 ,
1817     Keyword7 ,
1818     Keyword8 ,
1819     Keyword9 ,
1820     Name.Builtin ,
1821     Name.Class ,
1822     Name.Constructor ,
1823     Name.Decorator ,
1824     Name.Field ,
1825     Name.Function ,
1826     Name.Module ,
1827     Name.Namespace ,
1828     Name.Table ,
1829     Name.Type ,
1830     Number ,
1831     Number.Internal ,
1832     Operator ,
1833     Operator.Word ,
1834     Preproc ,
1835     Prompt ,
1836     String.Doc ,
1837     String.Doc.Internal ,
1838     String.Interpol ,
1839     String.Long ,

```

```

1840 String.Long.Internal ,
1841 String.Short ,
1842 String.Short.Internal ,
1843 Tag ,
1844 TypeParameter ,
1845 UserFunction ,

```

TypeExpression is an internal style for expressions which defines types in OCaml.

```

1846 TypeExpression ,

```

Now, specific styles for the languages created with \NewPitonLanguage with the syntax of listings.

```

1847 Directive
1848 }
1849 \clist_map_inline:Nn \g_@@_styles_clist
1850 {
1851   \keys_define:nn { piton / Styles }
1852   {
1853     #1 .value_required:n = true ,
1854     #1 .code:n =
1855       \tl_set:cn
1856       {
1857         pitonStyle _
1858         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1859         { \l_@@_SetPitonStyle_option_str _ }
1860         #1
1861       }
1862     { ##1 }
1863   }
1864 }
1865
1866 \keys_define:nn { piton / Styles }
1867 {
1868   String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1869   String      .value_required:n = true ,
1870   Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1871   Comment.Math .value_required:n = true ,
1872   unknown     .code:n = \@@_unknown_style:
1873 }

```

For the language `expl`, it's possible to create “on the fly” some styles of the form `Module.name` or `Type.name`. For the other languages, it's not possible.

```

1874 \cs_new_protected:Npn \@@_unknown_style:
1875 {
1876   \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1877   {
1878     \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1879     \seq_get_left:NN \l_tmpa_seq \l_tmpa_str

```

Now, the first part of the key (before the first period) is stored in `\l_tmpa_str`.

```

1880     \bool_lazy_and:nnTF
1881     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1882     {
1883       \str_if_eq_p:Vn \l_tmpa_str { Module }
1884       ||
1885       \str_if_eq_p:Vn \l_tmpa_str { Type }
1886     }

```

Now, we will create a new style.

```

1887     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1888     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1889   }
1890   { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1891 }

```

```

1892 \SetPitonStyle[OCaml]
1893 {
1894     TypeExpression =
1895     {
1896         \SetPitonStyle [ OCaml ] { Identifier = \PitonStyle { Name.Type } }
1897         \@@_piton:n
1898     }
1899 }

```

We add the word `String` to the list of the styles because we will use that list in the error message for an unknown key in `\SetPitonStyle`.

```

1900 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that clist.

```

1901 \clist_gsort:Nn \g_@@_styles_clist
1902 {
1903     \str_compare:nNnTF { #1 } < { #2 }
1904     \sort_return_same:
1905     \sort_return_swapped:
1906 }

```

```

1907 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1908
1909 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1910
1911 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1912 {
1913     \tl_set:Nn \l_tmpa_tl { #1 }

```

We have to begin by a substitution for the spaces. Otherwise, they would be gobbled in the `\tl_map_inline:Nn`.

```

1914     \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space
1915     \seq_clear:N \l_tmpa_seq
1916     \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1917     \seq_use:Nn \l_tmpa_seq { \- }
1918 }

```

```

1919 \cs_new_protected:Npn \@@_comment:n #1
1920 {
1921     \PitonStyle { Comment }
1922     {
1923         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1924         {
1925             \tl_set:Nn \l_tmpa_tl { #1 }
1926             \tl_replace_all:NVn \l_tmpa_tl
1927             \c_catcode_other_space_tl
1928             \@@_breakable_space:
1929             \l_tmpa_tl
1930         }
1931         { #1 }
1932     }
1933 }

```

```

1934 \cs_new_protected:Npn \@@_string_long:n #1
1935 {
1936     \PitonStyle { String.Long }
1937     {
1938         \bool_if:NTF \l_@@_break_strings_anywhere_bool
1939         { \@@_actually_break_anywhere:n { #1 } }
1940         {

```

We have, when `break-lines-in-Piton` is in force, to replace the spaces by `\@@_breakable_space:` because, when we have done a similar job in `\@@_replace_spaces:n` used in `\@@_begin_line:`, that job was not able to do the replacement in the brace group `{...}` of `\PitonStyle{String.Long}{...}` because we used a `\tl_replace_all:Nvn`. At that time, it would have been possible to use a `\tl_regex_replace_all:Nnn` but it is notoriously slow.

```

1941         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1942         {
1943             \tl_set:Nn \l_tmpa_tl { #1 }
1944             \tl_replace_all:Nvn \l_tmpa_tl
1945                 \c_catcode_other_space_tl
1946                 \@@_breakable_space:
1947             \l_tmpa_tl
1948         }
1949         { #1 }
1950     }
1951 }
1952 }
1953 \cs_new_protected:Npn \@@_string_short:n #1
1954 {
1955     \PitonStyle { String.Short }
1956     {
1957         \bool_if:NT \l_@@_break_strings_anywhere_bool
1958         { \@@_actually_break_anywhere:n }
1959         { #1 }
1960     }
1961 }
1962 \cs_new_protected:Npn \@@_string_doc:n #1
1963 {
1964     \PitonStyle { String.Doc }
1965     {
1966         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1967         {
1968             \tl_set:Nn \l_tmpa_tl { #1 }
1969             \tl_replace_all:Nvn \l_tmpa_tl
1970                 \c_catcode_other_space_tl
1971                 \@@_breakable_space:
1972             \l_tmpa_tl
1973         }
1974         { #1 }
1975     }
1976 }
1977 \cs_new_protected:Npn \@@_number:n #1
1978 {
1979     \PitonStyle { Number }
1980     {
1981         \bool_if:NT \l_@@_break_numbers_anywhere_bool
1982         { \@@_actually_break_anywhere:n }
1983         { #1 }
1984     }
1985 }

```

## 2.9 The initial styles

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

```

1986 \SetPitonStyle
1987 {
1988     Comment           = \color [ HTML ] { 0099FF } \itshape ,
1989     Comment.Internal  = \@@_comment:n ,
1990     Exception         = \color [ HTML ] { CC0000 } ,
1991     Keyword           = \color [ HTML ] { 006699 } \bfseries ,
1992     Keyword.Governing = \color [ HTML ] { 006699 } \bfseries ,

```

```

1993 Keyword.Constant      = \color [ HTML ] { 006699 } \bfseries ,
1994 Name.Builtin           = \color [ HTML ] { 336666 } ,
1995 Name.Decorator          = \color [ HTML ] { 9999FF } ,
1996 Name.Class              = \color [ HTML ] { 00AA88 } \bfseries ,
1997 Name.Function           = \color [ HTML ] { CC00FF } ,
1998 Name.Namespace          = \color [ HTML ] { 00CCFF } ,
1999 Name.Constructor        = \color [ HTML ] { 006000 } \bfseries ,
2000 Name.Field              = \color [ HTML ] { AA6600 } ,
2001 Name.Module             = \color [ HTML ] { 0060A0 } \bfseries ,
2002 Name.Table              = \color [ HTML ] { 309030 } ,
2003 Number                  = \color [ HTML ] { FF6600 } ,
2004 Number.Internal         = \@@_number:n ,
2005 Operator                = \color [ HTML ] { 555555 } ,
2006 Operator.Word           = \bfseries ,
2007 String                  = \color [ HTML ] { CC3300 } ,
2008 String.Long.Internal    = \@@_string_long:n ,
2009 String.Short.Internal   = \@@_string_short:n ,
2010 String.Doc.Internal     = \@@_string_doc:n ,
2011 String.Doc              = \color [ HTML ] { CC3300 } \itshape ,
2012 String.Interpol         = \color [ HTML ] { AA0000 } ,
2013 Comment.LaTeX           = \normalfont \color [ rgb ] { .468, .532, .6 } ,
2014 Name.Type               = \color [ HTML ] { 336666 } ,
2015 InitialValues           = \@@_piton:n ,
2016 Interpol.Inside         = { \l_@@_font_command_tl \@@_piton:n } ,
2017 TypeParameter           = \color [ HTML ] { 336666 } \itshape ,
2018 Preproc                 = \color [ HTML ] { AA6600 } \slshape ,

```

We need the command `\@@_identifier:n` because of the command `\SetPitonIdentifier`. The command `\@@_identifier:n` will potentially call the style `Identifier` (which is a user-style, not an internal style).

```

2019 Identifier.Internal    = \@@_identifier:n ,
2020 Identifier               = ,
2021 Directive               = \color [ HTML ] { AA6600 } ,
2022 Tag                     = \colorbox { gray!10 } ,
2023 UserFunction            = \PitonStyle { Identifier } ,
2024 Prompt                  = ,
2025 Discard                  = \use_none:n
2026 }

```

## 2.10 Styles specific to the language expl

```

2027 \clist_new:N \g_@@_expl_styles_clist
2028 \clist_gset:Nn \g_@@_expl_styles_clist
2029 {
2030   Scope.l ,
2031   Scope.g ,
2032   Scope.c
2033 }
2034 \clist_map_inline:Nn \g_@@_expl_styles_clist
2035 {
2036   \keys_define:nn { piton / Styles }
2037   {
2038     #1 .value_required:n = true ,
2039     #1 .code:n =
2040       \tl_set:cn
2041       {
2042         pitonStyle _
2043         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
2044         { \l_@@_SetPitonStyle_option_str _ }
2045         #1
2046       }
2047     { ##1 }

```

```

2048     }
2049 }
2050 \SetPitonStyle [ expl ]
2051 {
2052     Scope.l           = ,
2053     Scope.g           = \bfseries ,
2054     Scope.c           = \slshape ,
2055     Type.bool         = \color [ HTML ] { AA6600 } ,
2056     Type.box          = \color [ HTML ] { 267910 } ,
2057     Type.clist        = \color [ HTML ] { 309030 } ,
2058     Type.fp           = \color [ HTML ] { FF3300 } ,
2059     Type.int           = \color [ HTML ] { FF6600 } ,
2060     Type.seq          = \color [ HTML ] { 309030 } ,
2061     Type.skip         = \color [ HTML ] { 0CC060 } ,
2062     Type.str          = \color [ HTML ] { CC3300 } ,
2063     Type.tl           = \color [ HTML ] { AA2200 } ,
2064     Module.bool       = \color [ HTML ] { AA6600 } ,
2065     Module.box        = \color [ HTML ] { 267910 } ,
2066     Module.cs         = \bfseries \color [ HTML ] { 006699 } ,
2067     Module.exp        = \bfseries \color [ HTML ] { 404040 } ,
2068     Module.hbox       = \color [ HTML ] { 267910 } ,
2069     Module.prg        = \bfseries ,
2070     Module.clist      = \color [ HTML ] { 309030 } ,
2071     Module.fp         = \color [ HTML ] { FF3300 } ,
2072     Module.int        = \color [ HTML ] { FF6600 } ,
2073     Module.seq        = \color [ HTML ] { 309030 } ,
2074     Module.skip       = \color [ HTML ] { 0CC060 } ,
2075     Module.str        = \color [ HTML ] { CC3300 } ,
2076     Module.tl         = \color [ HTML ] { AA2200 } ,
2077     Module.vbox       = \color [ HTML ] { 267910 }
2078 }

```

If the key `math-comments` has been used in the preamble of the LaTeX document, 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)).

```

2079 \hook_gput_code:nnn { begindocument } { . }
2080 {
2081     \bool_if:NT \g_@@_math_comments_bool
2082     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
2083 }

```

## 2.11 Highlighting some identifiers

```

2084 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
2085 {
2086     \clist_set:Nn \l_tmpa_clist { #2 }
2087     \tl_if_novalue:nTF { #1 }
2088     {
2089         \clist_map_inline:Nn \l_tmpa_clist
2090         { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
2091     }
2092     {
2093         \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
2094         \str_if_eq:onT \l_tmpa_str { current-language }
2095         { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
2096         \clist_map_inline:Nn \l_tmpa_clist
2097         { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
2098     }
2099 }
2100 \cs_new_protected:Npn \@@_identifier:n #1
2101 {
2102     \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ #1 }

```

```

2103     {
2104         \cs_if_exist_use:cF { PitonIdentifier _ #1 }
2105         { \PitonStyle { Identifier } }
2106     }
2107     { #1 }
2108 }

```

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

```

2109 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
2110 {

```

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

```

2111     { \PitonStyle { Name.Function } { #1 } }

```

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

```

2112     \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ #1 }
2113     { \PitonStyle { UserFunction } }

```

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

```

2114     \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
2115     { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
2116     \seq_gput_right:cn { g_@@_functions _ \l_piton_language_str _ seq } { #1 }

```

We update `g_@@_languages_seq` which is used only by the command `\PitonClearUserFunctions` when it's used without its optional argument.

```

2117     \seq_if_in:NoF \g_@@_languages_seq { \l_piton_language_str }
2118     { \seq_gput_left:No \g_@@_languages_seq { \l_piton_language_str } }
2119 }

```

```

2120 \NewDocumentCommand \PitonClearUserFunctions { ! o }
2121 {
2122     \tl_if_novalue:nTF { #1 }

```

If the command is used without its optional argument, we will deleted the user language for all the computer languages.

```

2123         { \@@_clear_all_functions: }
2124         { \@@_clear_list_functions:n { #1 } }
2125     }

```

```

2126 \cs_new_protected:Npn \@@_clear_list_functions:n #1
2127 {
2128     \clist_set:Nn \l_tmpa_clist { #1 }
2129     \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
2130     \clist_map_inline:nn { #1 }
2131     { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
2132 }

```

```

2133 \cs_new_protected:Npn \@@_clear_functions_i:n #1
2134 { \@@_clear_functions_ii:n { \str_lowercase:n { #1 } } }

```

The following command clears the list of the user-defined functions for the language provided in argument (mandatory in lower case).

```

2135 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2136 {
2137     \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2138     {
2139         \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2140         { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }

```



```

2141     \seq_gclear:c { g_@@_functions _ #1 _ seq }
2142   }
2143 }
2144 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }

2145 \cs_new_protected:Npn \@@_clear_functions:n #1
2146 {
2147   \@@_clear_functions_i:n { #1 }
2148   \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2149 }

```

The following command clears all the user-defined functions for all the computer languages.

```

2150 \cs_new_protected:Npn \@@_clear_all_functions:
2151 {
2152   \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2153   \seq_gclear:N \g_@@_languages_seq
2154 }

```

```

2155 \AtEndDocument
2156 {

```

For the files written on the disk (with the key `write`), all the job is done by Lua.

```

2157   \lua_now:n { piton.write_files_now ( ) }

```

For the files joined in the PDF, we have a modern version which uses the package `pdfmanagement` of LaTeX and a legacy mechanism.

```

2158   \IfPDFManagementActiveTF
2159   { \@@_join_files: }
2160   { \@@_join_files_legacy: }
2161 }

```

If the new package `pdfmanagement` is used, we insert the file directly in the catalog of the PDF file.

```

2162 \cs_new_protected:Npn \@@_join_files:
2163 {
2164   \seq_map_inline:Nn \g_@@_join_seq
2165   {
2166     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2167     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2168     \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2169     {
2170       <<
2171         /Type /Filespec
2172         /UF <\l_tmpa_str>
2173         /EF << /F~\pdf_object_ref_last: >>
2174         /Desc (Computer~listing)
2175         /AFRelationship /Supplement
2176       >>
2177     }
2178   }
2179 }

```

The legacy version of `\@@_join_files:` will be used when the new package `pdfmanagement` is *not* used. In that case, we can't insert the file directly in the catalog of the pdf file. Therefore, we insert the file linked to an annotation in a page of the PDF file. We try to make the annotation itself invisible with several technics.

```

2180 \cs_new_protected:Npn \@@_join_files_legacy:
2181 {
2182   \seq_map_inline:Nn \g_@@_join_seq
2183   {
2184     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2185     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2186     \pdfextension annot~width~0pt~height~0pt~depth~0pt

```

The entry `/F` in the PDF dictionary of the annotation is an unsigned 32-bit integer containing flags specifying various characteristics of the annotation. The bit in position 2 means *Hidden*. However, despite that bit which means *Hidden*, some PDF readers show the annotation. That's why we have used `width Opt height Opt depth Opt`.

```

2187     {
2188         /Subtype /FileAttachment
2189         /F~2
2190         /Name /Paperclip
2191         /Contents (Computer~listing)
2192         /FS <<
2193             /Type /Filespec

```

We have previously converted the name of the embedded file in `utf16/hex` with the BOM of big endian and now we can write a PDF string between `<` and `>` (with that encoding).

```

2194         /UF <\l_tmpa_str>

```

It would have been possible to write `\pdffeedback lastobj~0~R` instead `\pdf_object_ref_last:` since LuaTeX is the only engine allowed by piton. Remark that `\pdf_object_ref_last:` is in the LaTeX kernel (not in the package `pdfmanagement`).

```

2195         /EF << /F~\pdf_object_ref_last: >>
2196         /AFRelationship /Supplement
2197     >>
2198 }
2199 }
2200 }

```

## 2.12 Spaces of indentation

```

2201 \cs_new_protected:Npn \@@_define_leading_space_normal:
2202 {
2203     \cs_set_protected:Npn \@@_leading_space:
2204     {
2205         \int_gincr:N \g_@@_indentation_int

```

Be careful: the `\hbox:n` is mandatory.

```

2206         \hbox:n { ~ }
2207     }
2208 }
2209 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2210 {
2211     \cs_set_protected:Npn \@@_leading_space:
2212     {
2213         \int_gincr:N \g_@@_indentation_int
2214         \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2215         {
2216             \color { white }
2217             \transparent { 0 }
2218             . % previously : □ U+2423
2219         }
2220         \pdfextension literal { EMC }
2221     }
2222 }
2223 \@@_define_leading_space_Foxit:

```

## 2.13 Security

```

2224 \AddToHook { env / piton / before }
2225 { \@@_fatal:n { No~environment~piton } }

```

## 2.14 The error messages of the package

When there is a unknown key, we try a “normal form” of the key and, when that normal form exists, we add that information in the error message.

The normal form is the lower case form of the key, with all the spaces replaced by hyphens (there is never spaces in the keys of piton).

#1 is a clist of names of sets of keys and #2 is the error message to send.

```

2226 \cs_new_protected:Npn \@@_unknown_key:nn #1 #2
2227 {
2228   \str_set_eq:NN \l_tmpa_str \l_keys_key_str
2229   \str_replace_all:Nnn \l_tmpa_str { ~ } { - }
2230   \str_set:Nx \l_tmpa_str { \str_lowercase:f { \l_tmpa_str } }
2231   \bool_set_false:N \l_tmpa_bool
2232   \clist_map_inline:nn { #1 }
2233   {
2234     \keys_if_exist:neT { ##1 } { \l_tmpa_str }
2235     {
2236       \@@_error:n { key~with~normal~form~exists }
2237       \bool_set_true:N \l_tmpa_bool
2238       \clist_map_break:
2239     }
2240   }
2241   \bool_if:NF \l_tmpa_bool { \@@_error:n { #2 } }
2242 }
2243 \@@_msg_new:nn { key~with~normal~form~exists }
2244 {
2245   The~key~'\l_keys_key_str'~does~not~exists.~It~will~be~ignored.\\
2246   Maybe~you~want~to~use~the~key~'\l_tmpa_str'.
2247 }
2248 \@@_msg_new:nn { No~environment~piton }
2249 {
2250   There~is~no~environment~piton!\\
2251   There~is~an~environment~{Piton}~and~a~command~
2252   \token_to_str:N \piton\ but~there~is~no~environment~
2253   {piton}.~This~error~is~fatal.
2254 }
2255 \@@_msg_new:nn { rounded~corners~without~Tikz }
2256 {
2257   TikZ~not~used \\
2258   You~can't~use~the~key~'rounded~corners'~because~
2259   you~have~not~loaded~the~package~TikZ. \\
2260   If~you~go~on,~that~key~will~be~ignored. \\
2261   You~won't~have~similar~error~till~the~end~of~the~document.
2262 }
2263 \@@_msg_new:nn { tcolorbox~not~loaded }
2264 {
2265   tcolorbox~not~loaded \\
2266   You~can't~use~the~key~'tcolorbox'~because~
2267   you~have~not~loaded~the~package~tcolorbox. \\
2268   Use~\token_to_str:N \usepackage[breakable]{tcolorbox}. \\
2269   If~you~go~on,~that~key~will~be~ignored.
2270 }
2271 \@@_msg_new:nn { library~breakable~not~loaded }
2272 {
2273   breakable~not~loaded \\
2274   You~can't~use~the~key~'tcolorbox'~because~
2275   you~have~not~loaded~the~library~'breakable'~of~tcolorbox'. \\
2276   Use~\token_to_str:N \tcbuselibrary{breakable}~in~the~preamble~
2277   of~your~document.\\
2278   If~you~go~on,~that~key~will~be~ignored.
2279 }
2280 \@@_msg_new:nn { Language~not~defined }
2281 {
2282   Language~not~defined \\
2283   The~language~'\l_tmpa_tl'~has~not~been~defined~previously.\\

```

```

2284     If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2285     will~be~ignored.
2286 }

2287 \@@_msg_new:nn { bad~version~of~piton.lua }
2288 {
2289     Bad~number~version~of~'piton.lua'\
2290     The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2291     version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2292     address~that~issue.
2293 }

2294 \@@_msg_new:nn { Unknown~key~NewPitonLanguage }
2295 {
2296     Unknown~key~for~\token_to_str:N \NewPitonLanguage.\
2297     The~key~'\l_keys_key_str'~is~unknown.\
2298     This~key~will~be~ignored.\
2299 }

2300 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
2301 {
2302     The~style~'\l_keys_key_str'~is~unknown.\
2303     This~setting~will~be~ignored.\
2304     The~available~styles~are~(in~alphabetic~order):~
2305     \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2306 }

2307 \@@_msg_new:nn { Invalid~key }
2308 {
2309     Wrong~use~of~key.\
2310     You~can't~use~the~key~'\l_keys_key_str'~here.\
2311     That~key~will~be~ignored.
2312 }

2313 \@@_msg_new:nn { Unknown~key~for~line~numbers }
2314 {
2315     Unknown~key. \
2316     The~key~'line~numbers / \l_keys_key_str'~is~unknown.\
2317     The~available~keys~of~the~family~'line~numbers'~are~(in~
2318     alphabetic~order):~
2319     absolute,~false,~label~empty~lines,~position,~resume,~skip~empty~lines,~
2320     sep,~start~and~true.\
2321     That~key~will~be~ignored.
2322 }

2323 \@@_msg_new:nn { Unknown~key~for~marker }
2324 {
2325     Unknown~key. \
2326     The~key~'marker / \l_keys_key_str'~is~unknown.\
2327     The~available~keys~of~the~family~'marker'~are~(in~
2328     alphabetic~order):~ beginning,~end~and~include~lines.\
2329     That~key~will~be~ignored.
2330 }

2331 \@@_msg_new:nn { bad~range~specification }
2332 {
2333     Incompatible~keys.\
2334     You~can't~specify~the~range~of~lines~to~include~by~using~both~
2335     markers~and~explicit~number~of~lines.\
2336     Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2337 }

2338 \cs_new_nopar:Nn \@@_thepage:
2339 {
2340     \thepage
2341     \cs_if_exist:NT \insertframenumber
2342     {
2343         ~(frame~\insertframenumber

```

```

2344     \cs_if_exist:NT \beamer@slidenummer { ,~slide~\insertslidenummer }
2345   )
2346 }
2347 }

```

We don't give the name `syntax error` for the following error because you should not give a name with a space because such space could be replaced by U+2423 when the key `show-spaces` is in force in the command `\piton`.

```

2348 \@@_msg_new:nn { SyntaxError }
2349 {
2350   Syntax~Error~on~page~\@@_thepage:.\
2351   Your~code~of~the~language~'\l_piton_language_str'~is~not~
2352   syntactically~correct.\
2353   It~won't~be~printed~in~the~PDF~file.
2354 }
2355 \@@_msg_new:nn { FileError }
2356 {
2357   File~Error.\
2358   It's~not~possible~to~write~on~the~file~'#1' \
2359   \sys_if_shell_unrestricted:F
2360   { (try~to~compile~with~'lualatex--shell-escape').\ }
2361   If~you~go~on,~nothing~will~be~written~on~that~file.
2362 }
2363 \@@_msg_new:nn { InexistentDirectory }
2364 {
2365   Inexistent~directory.\
2366   The~directory~'\l_@@_path_write_str'~
2367   given~in~the~key~'path-write'~does~not~exist.\
2368   Nothing~will~be~written~on~'\l_@@_write_str'.
2369 }
2370 \@@_msg_new:nn { begin~marker~not~found }
2371 {
2372   Marker~not~found.\
2373   The~range~'\l_@@_begin_range_str'~provided~to~the~
2374   command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
2375   The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
2376 }
2377 \@@_msg_new:nn { end~marker~not~found }
2378 {
2379   Marker~not~found.\
2380   The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
2381   provided~to~the~command~\token_to_str:N \PitonInputFile\
2382   has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
2383   be~inserted~till~the~end.
2384 }
2385 \@@_msg_new:nn { Unknown~file }
2386 {
2387   Unknown~file. \
2388   The~file~'#1'~is~unknown.\
2389   Your~command~\token_to_str:N \PitonInputFile\ will~be~discarded.
2390 }
2391 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2392 {
2393   \bool_if:NF \g_@@_beamer_bool
2394   { \@@_error_or_warning:n { Without~beamer } }
2395 }
2396 \@@_msg_new:nn { Without~beamer }
2397 {
2398   Key~'\l_keys_key_str'~without~Beamer.\
2399   You~should~not~use~the~key~'\l_keys_key_str'~since~you~
2400   are~not~in~Beamer.\

```

```

2401     However,~you~can~go~on.
2402 }
2403 \@@_msg_new:nn { rowcolor~in~detected-commands }
2404 {
2405     'rowcolor'~forbidden~in~'detected-commands'.\\
2406     You~should~put~'rowcolor'~in~'raw-detected-commands'.\\
2407     That~key~will~be~ignored.
2408 }
2409 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2410 {
2411     Unknown~key. \\
2412     The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
2413     It~will~be~ignored.\\
2414     For~a~list~of~the~available~keys,~type~H~<return>.
2415 }
2416 {
2417     The~available~keys~are~(in~alphabetic~order):~
2418     annotation,~
2419     add-to-split-separation,~
2420     auto-gobble,~
2421     background-color,~
2422     begin-range,~
2423     box,~
2424     break-lines,~
2425     break-lines-in-piton,~
2426     break-lines-in-Piton,~
2427     break-numbers-anywhere,~
2428     break-strings-anywhere,~
2429     continuation-symbol,~
2430     continuation-symbol-on-indentation,~
2431     detected-beamer-commands,~
2432     detected-beamer-environments,~
2433     detected-commands,~
2434     end-of-broken-line,~
2435     end-range,~
2436     env-gobble,~
2437     env-used-by-split,~
2438     font-command,~
2439     gobble,~
2440     indent-broken-lines,~
2441     join,~
2442     label-as-zlabel,~
2443     language,~
2444     left-margin,~
2445     line-numbers/,~
2446     marker/,~
2447     math-comments,~
2448     no-join,~
2449     no-write,~
2450     path,~
2451     path-write,~
2452     print,~
2453     prompt-background-color,~
2454     raw-detected-commands,~
2455     resume,~
2456     right-margin,~
2457     rounded-corners,~
2458     show-spaces,~
2459     show-spaces-in-strings,~
2460     splittable,~
2461     splittable-on-empty-lines,~
2462     split-on-empty-lines,~
2463     split-separation,~

```

```

2464     tabs-auto-gobble,~
2465     tab-size,~
2466     tcolorbox,~
2467     varwidth,~
2468     vertical-detected-commands,~
2469     width-and-write.
2470 }

2471 \@@_msg_new:nn { label-with-lines-numbers }
2472 {
2473     You~can't~use~the~command~\token_to_str:N \label\
2474     or~\token_to_str:N \zlabel\ because~the~key~'line-numbers'
2475     ~is~not~active.\
2476     If~you~go~on,~that~command~will~ignored.
2477 }

2478 \@@_msg_new:nn { overlay-without-beamer }
2479 {
2480     You~can't~use~an~argument~<...>~for~your~command~
2481     \token_to_str:N \PitonInputFile\ because~you~are~not~
2482     in~Beamer.\
2483     If~you~go~on,~that~argument~will~be~ignored.
2484 }

2485 \@@_msg_new:nn { label-as-zlabel-needs-zref-package }
2486 {
2487     The~key~'label-as-zlabel'~requires~the~package~'zref'.~
2488     Please~load~the~package~'zref'~before~setting~the~key.\
2489     This~error~is~fatal.
2490 }
2491 \hook_gput_code:nnn { begindocument } { . }
2492 {
2493     \bool_if:NT \g_@@_label_as_zlabel_bool
2494     {
2495         \IfPackageLoadedF { zref-base }
2496         { \@@_fatal:n { label-as-zlabel-needs-zref-package } }
2497     }
2498 }

```

## 2.15 We load piton.lua

```

2499 \cs_new_protected:Npn \@@_test_version:n #1
2500 {
2501     \str_if_eq:onF \PitonFileVersion { #1 }
2502     { \@@_error:n { bad~version~of~piton.lua } }
2503 }

2504 \hook_gput_code:nnn { begindocument } { . }
2505 {
2506     \lua_load_module:n { piton }
2507     \lua_now:n
2508     {
2509         tex.sprint ( luatexbase.catcodetables.expl ,
2510                     [[\@@_test_version:n {}] .. piton_version .. "]" )
2511     }
2512 }

```

</STY>

### 3 The Lua part of the implementation

The Lua code will be loaded via a `{luacode*}` environment. The environment is by itself a Lua block and the local declarations will be local to that block. All the global functions (used by the L3 parts of the implementation) will be put in a Lua table called `piton`.

```
2513  $\langle$ *LUA $\rangle$ 
2514 piton.comment_latex = piton.comment_latex or ">"
2515 piton.comment_latex = "#" .. piton.comment_latex
```

The table `piton.write_files` will contain the contents of all the files that we will write on the disk in the `\AtEndDocument` (if the user has used the key `write-file`). The table `piton.join_files` is similar for the key `join`.

```
2516 piton.write_files = { }
2517 piton.join_files = { }

2518 local sprintL3
2519 function sprintL3 ( s )
2520 tex.sprint ( luatexbase.catcodetables.expl , s )
2521 end
```

#### 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.

```
2522 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2523 local Cg, Cmt, Cb = lpeg.Cg, lpeg.Cmt, lpeg.Cb
2524 local B, R = lpeg.B, lpeg.R
```

The following line is mandatory.

```
2525 lpeg.locale(lpeg)
```

#### 3.2 The functions Q, K, WithStyle, etc.

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 computer listings that `piton` will typeset verbatim (thanks to the catcode “other”).

```
2526 local Q
2527 function Q ( pattern )
2528 return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2529 end
```

The function `L` takes in as argument a pattern and returns a LPEG *which does a capture* of the pattern. That capture will be sent to LaTeX with standard LaTeX catcodes for all the characters: the elements captured will be formatted as normal LaTeX codes. It's suitable for the “LaTeX comments” in the environments `{Piton}` and the elements between `begin-escape` and `end-escape`. That function won't be much used.

```
2530 local L
2531 function L ( pattern ) return
2532   Ct ( C ( pattern ) )
2533 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, unlike the previous one, will be widely used.

```

2534 local Lc
2535 function Lc ( string ) return
2536   Cc ( { luatexbase.catcodetables.expl , string } )
2537 end

```

The function `K` creates a LPEG which will return as capture the whole LaTeX code corresponding to a Python chunk (that is to say with the LaTeX formatting instructions corresponding to the syntactic nature of that Python chunk). The first argument is a Lua string corresponding to the name of a `piton` style and the second element is a pattern (that is to say a LPEG without capture)

```

2538 local K
2539 function K ( style , pattern ) return
2540   Lc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] )
2541   * Q ( pattern )
2542   * Lc "}" ]
2543 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 function `WithStyle` is similar to the function `K` but should be used for multi-lines elements.

```

2544 local WithStyle
2545 function WithStyle ( style , pattern ) return
2546   Ct ( Cc "Open" * Cc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] * Cc "}" ] )
2547   * pattern
2548   * Ct ( Cc "Close" )
2549 end

```

The following LPEG catches the Python chunks which are in LaTeX escapes (and that chunks will be considered as normal LaTeX constructions).

```

2550 Escape = P ( false )
2551 EscapeClean = P ( false )
2552 if piton.begin_escape then
2553   Escape =
2554     P ( piton.begin_escape )
2555     * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
2556     * P ( piton.end_escape )

```

The LPEG `EscapeClean` will be used in the LPEG `Clean` (and that LPEG is used to “clean” the code by removing the formatting elements).

```

2557   EscapeClean =
2558     P ( piton.begin_escape )
2559     * ( 1 - P ( piton.end_escape ) ) ^ 1
2560     * P ( piton.end_escape )
2561 end

2562 EscapeMath = P ( false )
2563 if piton.begin_escape_math then
2564   EscapeMath =
2565     P ( piton.begin_escape_math )
2566     * Lc "$"
2567     * L ( ( 1 - P ( piton.end_escape_math ) ) ^ 1 )
2568     * Lc "$"
2569     * P ( piton.end_escape_math )
2570 end

```

## The basic syntactic LPEG

```
2571 local alpha , digit = lpeg.alpha , lpeg.digit
2572 local space = P " "
```

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

```
2573 local letter = alpha + "_" + "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "î"
2574               + "ô" + "û" + "ü" + "Å" + "Ä" + "Ç" + "É" + "È" + "Ê" + "Ë"
2575               + "Ī" + "Ĭ" + "Ō" + "Ū" + "Ū"
2576
2577 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).

```
2578 local identifier = letter * alphanum ^ 0
```

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

```
2579 local Identifier = K ( 'Identifier.Internal' , identifier )
```

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

The following functions allow to recognize numbers that contains `_` among their digits, for example `1_000_000`, but also floating point numbers, numbers with exponents and numbers with different bases.<sup>3</sup>

```
2580 local allow_underscores_except_first
2581 function allow_underscores_except_first ( p )
2582     return p * (P "_" + p)^0
2583 end
2584 local allow_underscores
2585 function allow_underscores ( p )
2586     return (P "_" + p)^0
2587 end
2588 local digits_to_number
2589 function digits_to_number(prefix, digits)
2590     -- The edge cases of what is allowed in number literals is modelled after
2591     -- OCaml numbers, which seems to be the most permissive language
2592     -- in this regard (among C, OCaml, Python & SQL).
2593     return prefix
2594         * allow_underscores_except_first(digits^1)
2595         * (P "." * #(1 - P ".") * allow_underscores(digits))^~1
2596         * (S "eE" * S "+-~"^-1 * allow_underscores_except_first(digits^1))^~1
2597 end
```

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).

```
2598 local Number =
2599     K ( 'Number.Internal' ,
2600         digits_to_number (P "0x" + P "0X", R "af" + R "AF" + digit)
2601         + digits_to_number (P "0o" + P "0O", R "07")
2602         + digits_to_number (P "0b" + P "0B", R "01")
2603         + digits_to_number ( "" , digit )
2604     )
```

---

<sup>3</sup>The edge cases such as

We will now define the LPEG Word.

We have a problem in the following LPEG because, obviously, we should adjust the list of symbols with the delimiters of the current language (no?).

```
2605 local lpeg_central = 1 - S " '\r[{}]" - digit
```

We recall that `piton.begin_escape` and `piton.end_escape` are Lua strings corresponding to the keys `begin-escape` and `end-escape`.

```
2606 if piton.begin_escape then
2607   lpeg_central = lpeg_central - piton.begin_escape
2608 end
2609 if piton.begin_escape_math then
2610   lpeg_central = lpeg_central - piton.begin_escape_math
2611 end
2612 local Word = Q ( lpeg_central ^ 1 )
```

```
2613 local Space = Q " " ^ 1
2614 local SkipSpace = Q " " ^ 0
2615
2616 local Punct = Q ( S ".,:;!" )
2617
2618 local Tab = "\t" * Lc [[ \@_tab: ]]
```

```
2619 local LeadingSpace = Lc [[ \@_leading_space: ]] * P " "
```

```
2620 local Delim = Q ( S "[{}]" )
```

The following LPEG catches a space (U+0020) and replaces it by `\l_@@_space_in_string_t1`. It will be used in the strings. Usually, `\l_@@_space_in_string_t1` will contain a space and therefore there won't be any difference. However, when the key `show-spaces-in-strings` is in force, `\l_@@_space_in_string_t1` will contain `␣` (U+2423) in order to visualize the spaces.

```
2621 local SpaceInString = space * Lc [[ \l_@@_space_in_string_t1 ]]
```

### 3.3 The option 'detected-commands' and al.

We create four Lua tables called `detected_commands`, `raw_detected_commands`, `beamer_commands` and `beamer_environments`.

On the TeX side, the corresponding data have first been stored as clists.

Then, in a `\AtBeginDocument`, they have been converted in “toks registers” of TeX.

Now, on the Lua side, we are able to access to those “toks registers” with the special pseudo-table `tex.toks` of LuaTeX.

Remark that we can safely use `explode('')` to convert such “toks registers” in Lua tables since, in a clist of L3, there is no empty component and, for each component, there is no space on both sides (the `explode` of the Lua of LuaTeX is unable to do itself such purification of the components).

```
2622 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
2623 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
2624 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
2625 local beamer_environments = tex.toks.PitonBeamerEnvironments : explode ( ',' )
```

We will also create some LPEG.

According to our conventions, a LPEG with a name in camelCase is a LPEG which doesn't do any capture.

```
2626 local detectedCommands = P ( false )
2627 for _ , x in ipairs ( detected_commands ) do
2628   detectedCommands = detectedCommands + P ( "\\\" .. x )
2629 end
```

Further, we will have a LPEG called `DetectedCommands` (in PascalCase) which will be a LPEG *with* captures.

```

2630 local rawDetectedCommands = P ( false )
2631 for _ , x in ipairs ( raw_detected_commands ) do
2632   rawDetectedCommands = rawDetectedCommands + P ( "\\\" .. x )
2633 end
2634
2634 local beamerCommands = P ( false )
2635 for _ , x in ipairs ( beamer_commands ) do
2636   beamerCommands = beamerCommands + P ( "\\\" .. x )
2637 end
2638
2638 local beamerEnvironments = P ( false )
2639 for _ , x in ipairs ( beamer_environments ) do
2640   beamerEnvironments = beamerEnvironments + P ( x )
2641 end

```

### Several tools for the construction of the main LPEG

```

2642 local LPEG0 = { }
2643 local LPEG1 = { }
2644 local LPEG2 = { }
2645 local LPEG_cleaner = { }

```

For each language, we will need a pattern to match expressions with balanced braces. Those balanced braces must *not* take into account the braces present in strings of the language. However, the syntax for the strings is language-dependent. That's why we write a Lua function `Compute_braces` which will compute the pattern by taking in as argument a pattern for the strings of the language (at least the shorts strings). The argument of `Compute_braces` must be a pattern *which does no captures*.

```

2646 local Compute_braces
2647 function Compute_braces ( lpeg_string ) return
2648   P { "E" ,
2649     E =
2650       (
2651         "{" * V "E" * "}"
2652         +
2653         lpeg_string
2654         +
2655         ( 1 - S "{" )
2656         ) ^ 0
2657   }
2658 end

```

The following Lua function will compute the lpeg `DetectedCommands` which is a LPEG with captures.

```

2659 local Compute_DetectedCommands
2660 function Compute_DetectedCommands ( lang , braces ) return
2661   Ct (
2662     Cc "Open"
2663     * C ( detectedCommands * space ^ 0 * P "{" )
2664     * Cc "}"
2665   )
2666   * ( braces
2667     / ( function ( s )
2668         if s ~= '' then return
2669           LPEG1[lang] : match ( s )
2670         end
2671       end )
2672   )
2673   * P "}"
2674   * Ct ( Cc "Close" )
2675 end

```

```

2676 local Compute_RawDetectedCommands
2677 function Compute_RawDetectedCommands ( lang , braces ) return
2678   Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
2679 end

2680 local Compute_LPEG_cleaner
2681 function Compute_LPEG_cleaner ( lang , braces ) return
2682   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
2683         * ( braces
2684           / ( function ( s )
2685               if s ~= '' then return
2686                 LPEG_cleaner[lang] : match ( s )
2687             end
2688           end )
2689         )
2690         * "}"
2691         + EscapeClean
2692         + C ( P ( 1 ) )
2693         ) ^ 0 ) / table.concat
2694 end

```

The following function `ParseAgain` will be used in the definitions of the LPEG of the different computer languages when we will need to *parse again* a small chunk of code. It's a way to avoid the use of a actual *grammar* of LPEG (in a sens, a recursive regular expression).

Remark that there is no `piton` style associated to a chunk of code which is analyzed by `ParseAgain`. If we wish a `piton` style available to the end user (if he wish to format that element with a uniform font instead of an analyze by `ParseAgain`), we have to use `\@@_piton:n`.

```

2695 local ParseAgain
2696 function ParseAgain ( code )
2697   if code ~= '' then return

```

The variable `piton.language` is set in the function `piton.Parse`.

```

2698   LPEG1[piton.language] : match ( code )
2699   end
2700 end

```

**Constructions for Beamer** If the class `Beamer` is used, some environments and commands of `Beamer` are automatically detected in the listings of `piton`.

```

2701 local Beamer = P ( false )

```

The following Lua function will be used to compute the LPEG `Beamer` for each computer language. According to our conventions, the LPEG `Beamer`, with its name in PascalCase does captures.

```

2702 local Compute_Beamer
2703 function Compute_Beamer ( lang , braces )

```

We will compute in `lpeg` the LPEG that we will return.

```

2704 local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
2705 lpeg = lpeg +
2706   Ct ( Cc "Open"
2707         * C ( beamerCommands
2708             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2709             * P "{"
2710           )
2711         * Cc "}"
2712       )
2713   * ( braces /
2714     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2715   * "}"
2716   * Ct ( Cc "Close" )

```

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

```

2717 lpeg = lpeg +
2718   L ( P [[\alt]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2719   * ( braces /
2720     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2721   * L ( P "}" )
2722   * ( braces /
2723     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2724   * L ( P "}" )

```

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

```

2725 lpeg = lpeg +
2726   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2727   * ( braces
2728     / ( function ( s )
2729       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2730   * L ( P "}" )
2731   * ( braces
2732     / ( function ( s )
2733       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2734   * L ( P "}" )
2735   * ( braces
2736     / ( function ( s )
2737       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2738   * L ( P "}" )

```

Now, the environments of Beamer.

```

2739 for _ , x in ipairs ( beamer_environments ) do
2740   lpeg = lpeg +
2741     Ct ( Cc "Open"
2742       * C (
2743         P ( [[\begin{]] .. x .. "]" )
2744         * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2745       )
2746       * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
2747       * Cc ( [[\end{]] .. x .. "]" )
2748     )
2749   * (

```

We catch all the content of the Beamer environment which a LPEG which is a grammar because there may be nested environments of the same type (added 2025/11/14).

```

2750     (
2751       P { "E" ,
2752         E = (
2753           P ( [[\begin{]] .. x .. "]" )
2754           * V "E"
2755           * P ( [[\end{]] .. x .. "]" )
2756         +
2757         (
2758           1
2759           - P ( [[\begin{]] .. x .. "]" )
2760           - P ( [[\end{]] .. x .. "]" )
2761         )
2762         ) ^ 0
2763       }
2764     )
2765     / ( function ( s )
2766       if s ~= '' then return
2767         LPEG1[lang] : match ( s )
2768       end
2769     end )
2770   )

```

```

2771         * P ( [[\end{}} .. x .. "}" )
2772         * Ct ( Cc "Close" )
2773     end

```

Now, you can return the value we have computed.

```

2774     return lpeg
2775 end

```

The following LPEG is in relation with the key `math-comments`. It will be used in all the languages.

```

2776 local CommentMath =
2777     P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$" -- $

```

**EOL** There may be empty lines in the transcription of the prompt, *id est* lines of the form ... without space after and that's why we need `P " " ^ -1` with the `^ -1`.

```

2778 local Prompt =
2779     K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
2780     * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

2781 local EOL =
2782     P "\r"
2783     *
2784     (
2785         space ^ 0 * -1
2786         +
2787         Cc "EOL"
2788     )
2789     * ( LeadingSpace ^ 0 * # ( 1 - S " \r" ) ) ^ -1

```

The following LPEG `CommentLaTeX` is for what is called in that document the “LaTeX comments”.

```

2790 local CommentLaTeX =
2791     P ( piton.comment_latex )
2792     * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}}]
2793     * L ( ( 1 - P "\r" ) ^ 0 )
2794     * Lc "}}"
2795     * ( EOL + -1 )

```

### 3.4 The language Python

We open a Lua local scope for the language Python (of course, there will be also global definitions).

```

2796 --python Python
2797 do

```

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

```

2798 local Operator =
2799     K ( 'Operator' ,
2800         P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "://" + "**"
2801         + S "--+/*%=<>&.@|" )
2802
2803 local OperatorWord =
2804     K ( 'Operator.Word' , P "in" + "is" + "and" + "or" + "not" )

```

The keyword `in` in a construction such as “`for i in range(n)`” must be formatted as a keyword and not as an `Operator.Word` and that’s why we write the following LPEG `For`.

```

2805 local For = K ( 'Keyword' , P "for" )
2806         * Space
2807         * Identifier
2808         * Space
2809         * K ( 'Keyword' , P "in" )
2810
2811 local Keyword =
2812   K ( 'Keyword' ,
2813     P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
2814     "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
2815     "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
2816     "try" + "while" + "with" + "yield" + "yield from" )
2817   + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
2818
2819 local Builtin =
2820   K ( 'Name.Builtin' ,
2821     P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
2822     "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
2823     "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
2824     "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
2825     "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
2826     "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
2827     + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
2828     "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
2829     "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
2830     "vars" + "zip" )
2831
2832 local Exception =
2833   K ( 'Exception' ,
2834     P "ArithmeticError" + "AssertionError" + "AttributeError" +
2835     "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
2836     "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
2837     "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
2838     "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
2839     "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
2840     "NotImplementedError" + "OSError" + "OverflowError" +
2841     "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
2842     "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
2843     "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
2844     + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
2845     "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
2846     "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
2847     "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
2848     "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
2849     "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
2850     "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
2851     "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
2852     "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
2853     "RecursionError" )
2854
2855 local RaiseException = K ( 'Keyword' , P "raise" ) * SkipSpace * Exception * Q "("

```

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

```

2856 local Decorator = K ( 'Name.Decorator' , P "@" * letter ^ 1 )

```

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

Example: `class myclass:`



```

2857 local DefClass =
2858     K ( 'Keyword' , "class" ) * Space * K ( 'Name.Class' , identifier )

```

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

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

Example: `import numpy as np`

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

Example: `import math, numpy`

```

2859 local ImportAs =
2860     K ( 'Keyword' , "import" )
2861     * Space
2862     * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
2863     * (
2864         ( Space * K ( 'Keyword' , "as" ) * Space
2865           * K ( 'Name.Namespace' , identifier ) )
2866         +
2867         ( SkipSpace * Q "," * SkipSpace
2868           * K ( 'Name.Namespace' , identifier ) ) ^ 0
2869     )

```

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`

```

2870 local FromImport =
2871     K ( 'Keyword' , "from" )
2872     * Space * K ( 'Name.Namespace' , identifier )
2873     * Space * K ( 'Keyword' , "import" )

```

**The strings of Python** For the strings in Python, there are four categories of delimiters (without counting the prefixes for f-strings and raw strings). We will use, in the names of our LPEG, prefixes to distinguish the LPEG dealing with that categories of strings, as presented in the following tabular.

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

We have also to deal with the interpolations in the f-strings. Here is an example of a f-string with an interpolation and a format instruction<sup>4</sup> in that interpolation:

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

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

```

2874 local PercentInterpol =
2875     K ( 'String.Interpol' ,
2876         P "%"

```

<sup>4</sup>There is no special `piton` style for the formatting instruction (after the colon): the style which will be applied will be the style of the encompassing string, that is to say `String.Short` or `String.Long`.

```

2877 * ( "(" * alphanum ^ 1 * ")" ) ^ -1
2878 * ( S "-#0 +" ) ^ 0
2879 * ( digit ^ 1 + "*" ) ^ -1
2880 * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
2881 * ( S "HLL" ) ^ -1
2882 * S "sdfFeExXorgiGauc%"
2883 )

```

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>5</sup>

```

2884 local SingleShortString =
2885   WithStyle ( 'String.Short.Internal' ,

```

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

```

2886   Q ( P "f'" + "F'" )
2887   * (
2888     K ( 'String.Interpol' , "{" )
2889     * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0 )
2890     * Q ( P ":" * ( 1 - S "}':" ) ^ 0 ) ^ -1
2891     * K ( 'String.Interpol' , "}" )
2892     +
2893     SpaceInString
2894     +
2895     Q ( ( P "\\'" + "\\\\" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
2896   ) ^ 0
2897   * Q ""
2898 +

```

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

```

2899   Q ( P "'" + "r'" + "R'" )
2900   * ( Q ( ( P "\\'" + "\\\\" + 1 - S " 'r%" ) ^ 1 )
2901     + SpaceInString
2902     + PercentInterpol
2903     + Q "%"
2904   ) ^ 0
2905   * Q "" )

2906 local DoubleShortString =
2907   WithStyle ( 'String.Short.Internal' ,
2908     Q ( P "f\"" + "F\"" )
2909     * (
2910       K ( 'String.Interpol' , "{" )
2911       * K ( 'Interpol.Inside' , ( 1 - S "}\"::" ) ^ 0 )
2912       * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}\"::" ) ^ 0 ) ) ^ -1
2913       * K ( 'String.Interpol' , "}" )
2914       +
2915       SpaceInString
2916       +
2917       Q ( ( P "\\\"" + "\\\\" + "{{" + "}}" + 1 - S " {}\" ) ^ 1 )
2918     ) ^ 0
2919     * Q "\""
2920   +
2921   Q ( P "\" + "r\"" + "R\"" )
2922   * ( Q ( ( P "\\\"" + "\\\\" + 1 - S " \"r%" ) ^ 1 )
2923     + SpaceInString
2924     + PercentInterpol
2925     + Q "%"
2926   ) ^ 0
2927   * Q "\" )

```

---

<sup>5</sup>The interpolations are formatted with the `piton` style `Interpol.Inside`. The initial value of that style is `\\@@_piton:n` which means that the interpolations are parsed once again by `piton`.

```

2928
2929 local ShortString = SingleShortString + DoubleShortString

```

**Beamer** The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

2930 local braces =
2931   Compute_braces
2932   (
2933     ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
2934     * ( P '\\\"' + 1 - S "\"" ) ^ 0 * "\""
2935   +
2936     ( P '\'' + 'r\'' + 'R\'' + 'f\'' + 'F\'' )
2937     * ( P '\\\'' + 1 - S '\'' ) ^ 0 * '\''
2938   )
2939
2940 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

## Detected commands

```

2941 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
2942 + Compute_RawDetectedCommands ( 'python' , braces )

```

## LPEG\_cleaner

```

2943 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

## The long strings

```

2944 local SingleLongString =
2945   WithStyle ( 'String.Long.Internal' ,
2946     ( Q ( S "fF" * P "'''" )
2947       * (
2948         K ( 'String.Interpol' , "{" )
2949         * K ( 'Interpol.Inside' , ( 1 - S "};\r" - "'''" ) ^ 0 )
2950         * Q ( P ":" * ( 1 - S "};\r" - "'''" ) ^ 0 ) ^ -1
2951         * K ( 'String.Interpol' , "}" )
2952       +
2953       Q ( ( 1 - P "'''" - S "{'}\r" ) ^ 1 )
2954       +
2955       EOL
2956     ) ^ 0
2957   +
2958   Q ( ( S "rR" ) ^ -1 * "'''" )
2959   * (
2960     Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
2961     +
2962     PercentInterpol
2963     +
2964     P "%"
2965     +
2966     EOL
2967   ) ^ 0
2968 )
2969 * Q "'''" )

```

```

2970 local DoubleLongString =
2971   WithStyle ( 'String.Long.Internal' ,
2972     (
2973       Q ( S "fF" * "\"\\\"" )
2974       * (
2975         K ( 'String.Interpol', "{ " )
2976         * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "\"\\\"" ) ^ 0 )
2977         * Q ( ":" * (1 - S "}:\\r" - "\"\\\"" ) ^ 0 ) ^ -1
2978         * K ( 'String.Interpol' , "}" )
2979         +
2980         Q ( ( 1 - S "{}\\r" - "\"\\\"" ) ^ 1 )
2981         +
2982         EOL
2983       ) ^ 0
2984     +
2985     Q ( S "rR" ^ -1 * "\"\\\"" )
2986     * (
2987       Q ( ( 1 - P "\"\\\"" - S "%\\r" ) ^ 1 )
2988       +
2989       PercentInterpol
2990       +
2991       P "%"
2992       +
2993       EOL
2994     ) ^ 0
2995   )
2996   * Q "\"\\\""
2997 )
2998 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`).

```

2999 local StringDoc =
3000   K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\\\"" )
3001   * ( K ( 'String.Doc.Internal' , (1 - P "\"\\\"" - "\\r" ) ^ 0 ) * EOL
3002     * Tab ^ 0
3003   ) ^ 0
3004   * K ( 'String.Doc.Internal' , ( 1 - P "\"\\\"" - "\\r" ) ^ 0 * "\"\\\"" )

```

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

```

3005 local Comment =
3006   WithStyle
3007   ( 'Comment.Internal' ,
3008     Q "#" * ( CommentMath + Q ( ( 1 - S "$\\r" ) ^ 1 ) ) ^ 0 -- $
3009   )
3010   * ( 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).

```

3011 local expression =
3012   P { "E" ,
3013     E = ( "'" * ( P "\\'" + 1 - S "'\\r" ) ^ 0 * "'"
3014       + "\\\"" * ( P "\\\"" + 1 - S "\\\"\\r" ) ^ 0 * "\\\""
3015       + "{" * V "F" * "}"
3016       + "(" * V "F" * ")" )

```

```

3017         + "[" * V "F" * "]"
3018         + ( 1 - S "{ } ( [ ] \r , " ) ) ^ 0 ,
3019     F = (   "{" * V "F" * "}"
3020           + "(" * V "F" * ")"
3021           + "[" * V "F" * "]"
3022           + ( 1 - S "{ } ( [ ] \r \'"' " ) ) ^ 0
3023     }

```

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`.

```

3024     local Params =
3025     P { "E" ,
3026         E = ( V "F" * ( Q " , " * V "F" ) ^ 0 ) ^ -1 ,
3027         F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
3028           * ( Q ":" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3029           * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
3030     }

```

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`...

```

3031     local DefFunction =
3032     K ( 'Keyword' , "def" )
3033     * Space
3034     * K ( 'Name.Function.Internal' , identifier )
3035     * SkipSpace
3036     * Q "(" * Params * Q ")"
3037     * SkipSpace
3038     * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3039     * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
3040     * Q ":"
3041     * ( SkipSpace
3042       * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
3043       * Tab ^ 0
3044       * SkipSpace
3045       * StringDoc ^ 0 -- there may be additional docstrings
3046     ) ^ -1

```

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

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

## Miscellaneous

```

3047     local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

## The main LPEG for the language Python

```
3048 local EndKeyword
3049     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3050     EscapeMath + -1
```

First, the main loop :

```
3051 local Main =
3052     space ^ 0 * EOL -- faut-il le mettre en commentaire ?
3053     + Space
3054     + Tab
3055     + Escape + EscapeMath
3056     + Beamer
3057     + CommentLaTeX
3058     + DetectedCommands
3059     + Prompt
3060     + LongString
3061     + Comment
3062     + ExceptionInConsole
3063     + Delim
3064     + Operator
3065     + OperatorWord * EndKeyword
3066     + ShortString
3067     + Punct
3068     + FromImport
3069     + RaiseException
3070     + DefFunction
3071     + DefClass
3072     + For
3073     + Keyword * EndKeyword
3074     + Decorator
3075     + Builtin * EndKeyword
3076     + Identifier
3077     + Number
3078     + Word
```

Here, we must not put local, of course.

```
3079 LPEG1.python = Main ^ 0
```

We recall that each line in the Python code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>6</sup>.

```
3080 LPEG2.python =
3081     Ct (
3082         ( space ^ 0 * "\r" ) ^ -1
3083         * Lc [[ \@@_begin_line: ]]
3084         * LeadingSpace ^ 0
3085         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3086         * -1
3087         * Lc [[ \@@_end_line: ]]
3088     )
```

End of the Lua scope for the language Python.

```
3089 end
```

---

<sup>6</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:`

### 3.5 The language OCaml

We open a Lua local scope for the language OCaml (of course, there will be also global definitions).

```

3090 --ocaml Ocaml OCaml
3091 do

3092     local SkipSpace = ( Q " " + EOL ) ^ 0
3093     local Space = ( Q " " + EOL ) ^ 1

3094     local braces = Compute_braces ( '\"' * ( 1 - S "\"" ) ^ 0 * '\"' )

3095     if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
3096     DetectedCommands =
3097         Compute_DetectedCommands ( 'ocaml' , braces )
3098         + Compute_RawDetectedCommands ( 'ocaml' , braces )
3099     local Q

```

Usually, the following version of the function Q will be used without the second argument (**strict**), that is to say in a looser way. However, in some circumstances, we will need the “strict” version, for instance in DefFunction.

```

3100     function Q ( pattern, strict )
3101         if strict ~= nil then
3102             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3103         else
3104             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3105                 + Beamer + DetectedCommands + EscapeMath + Escape
3106         end
3107     end

3108     local K
3109     function K ( style , pattern, strict ) return
3110         Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
3111         * Q ( pattern, strict )
3112         * Lc "}"
3113     end

3114     local WithStyle
3115     function WithStyle ( style , pattern ) return
3116         Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{ ]] .. style .. "}{" ) * Cc "}" )
3117         * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
3118         * Ct ( Cc "Close" )
3119     end

```

The following LPEG corresponds to the balanced expressions (balanced according to the parenthesis). Of course, we must write (1 - S "(") with outer parenthesis.

```

3120     local balanced_parens =
3121         P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }

```

#### The strings of OCaml

```

3122     local ocaml_string =
3123         P "\""
3124         * (
3125             P " "
3126             +
3127             P ( ( 1 - S "\r" ) ^ 1 )
3128             +
3129             EOL -- ?
3130         ) ^ 0
3131     * P "\""

```

```

3132 local String =
3133   WithStyle
3134     ( 'String.Long.Internal' ,
3135       Q "\""
3136       * (
3137         SpaceInString
3138         +
3139         Q ( ( 1 - S " \"\r" ) ^ 1 )
3140         +
3141         EOL
3142       ) ^ 0
3143       * Q "\""
3144     )

```

Now, the “quoted strings” of OCaml (for example {`ext`|`Essai`|`ext`}).

For those strings, we will do two consecutive analysis. First an analysis to determine the whole string and, then, an analysis for the potential visual spaces and the EOL in the string.

The first analysis require a match-time capture. For explanations about that programmation, see the paragraphe *Lua’s long strings* in [www.inf.puc-rio.br/~roberto/lpeg](http://www.inf.puc-rio.br/~roberto/lpeg).

```

3145 local ext = ( R "az" + "_" ) ^ 0
3146 local open = "{" * Cg ( ext , 'init' ) * "/"
3147 local close = "/" * C ( ext ) * "}"
3148 local closeeq =
3149   Cmt ( close * Cb ( 'init' ) ,
3150     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3151 local QuotedStringBis =
3152   WithStyle ( 'String.Long.Internal' ,
3153     (
3154       Space
3155       +
3156       Q ( ( 1 - S " \r" ) ^ 1 )
3157       +
3158       EOL
3159     ) ^ 0 )

```

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

```

3160 local QuotedString =
3161   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3162   ( function ( s ) return QuotedStringBis : match ( s ) end )

```

In OCaml, the delimiters for the comments are (\* and \*). There are unsymmetrical and OCaml allows those comments to be nested. That’s why we need a grammar.

In these comments, we embed the math comments (between \$ and \$) and we embed also a treatment for the end of lines (since the comments may be multi-lines).

```

3163 local comment =
3164   P {
3165     "A" ,
3166     A = Q "(*"
3167       * ( V "A"
3168         + Q ( ( 1 - S "\r$\\"" - "(*" - "*)" ) ^ 1 ) -- $
3169         + ocaml_string
3170         + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3171         + EOL
3172       ) ^ 0
3173       * Q "*)"
3174   }
3175 local Comment = WithStyle ( 'Comment.Internal' , comment )

```



## Some standard LPEG

```
3176 local Delim = Q ( P "[" + "]" + S "[]" )
3177 local Punct = Q ( S ",:;! " )
```

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

```
3178 local cap_identifier = R "AZ" * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
```

We consider `::` and `[]` as constructors (of the lists) as does the Tuareg mode of Emacs.

```
3179 local Constructor =
3180   P "::"
```

Don't use `\hspace` instead of `\kern`

```
3181 * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3182 +
3183 P "[]"
3184 * Lc ([{\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]])
3185 K ( 'Name.Constructor' ,
3186     Q "`" ^ -1 * cap_identifier
3187     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )
```

```
3188 local ModuleType = K ( 'Name.Type' , cap_identifier )
```

```
3189 local OperatorWord =
3190   K ( 'Operator.Word' ,
3191       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )
```

In OCaml, some keywords are considered as *governing keywords* with some special syntactic characteristics.

```
3192 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3193   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3194   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3195   "struct" + "type" + "val"
```

```
3196 local Keyword =
3197   K ( 'Keyword' ,
3198       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3199       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3200       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3201       + "virtual" + "when" + "while" + "with" )
3202   + K ( 'Keyword.Constant' , P "true" + "false" )
3203   + K ( 'Keyword.Governing' , governing_keyword )
```

```
3204 local EndKeyword
3205   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3206   + EscapeMath + -1
```

Now, the identifier. Recall that we have also a LPEG `cap_identifier` for the identifiers beginning with a capital letter.

```
3207 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3208   - ( OperatorWord + Keyword ) * EndKeyword
```

We have the internal style `Identifier.Internal` in order to be able to implement the mechanism `\SetPitonIdentifier`. The final user has access to a style called `Identifier`.

```
3209 local Identifier = K ( 'Identifier.Internal' , identifier )
```

In OCaml, *character* is a type different of the type `string`.

```

3210 local ocaml_char =
3211   P "'" *
3212   (
3213     ( 1 - S "\\\" )
3214     + "\\\"
3215     * ( S "\\ntbr \"\"
3216         + digit * digit * digit
3217         + P "x" * ( digit + R "af" + R "AF" )
3218                   * ( digit + R "af" + R "AF" )
3219                   * ( digit + R "af" + R "AF" )
3220         + P "o" * R "03" * R "07" * R "07" )
3221   )
3222   * "'"
3223 local Char =
3224   K ( 'String.Short.Internal', ocaml_char )

```

For the parameter of the types (for example : ``\a` as in ``a` list).

```

3225 local TypeParameter =
3226   K ( 'TypeParameter' ,
3227       "'" * Q "_" ^ -1 * alpha ^ 1 * digit ^ 0 * ( # ( 1 - P "'" ) + -1 ) )

```

**DotNotation** Now, we deal with the notations with points (eg: `List.length`). In OCaml, such notation is used for the fields of the records and for the modules.

```

3228 local DotNotation =
3229   (
3230     K ( 'Name.Module' , cap_identifier )
3231     * Q "."
3232     * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3233     +
3234     Identifier
3235     * Q "."
3236     * K ( 'Name.Field' , identifier )
3237   )
3238   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

```

## The records

```

3239 local expression_for_fields_type =
3240   P { "E" ,
3241       E = ( "{" * V "F" * "}"
3242            + "(" * V "F" * ")"
3243            + TypeParameter
3244            + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
3245       F = ( "{" * V "F" * "}"
3246            + "(" * V "F" * ")"
3247            + ( 1 - S "{}()[]\r\"" ) + TypeParameter ) ^ 0
3248   }

```

```

3249 local expression_for_fields_value =
3250   P { "E" ,
3251       E = ( "{" * V "F" * "}"
3252            + "(" * V "F" * ")"
3253            + "[" * V "F" * "]"
3254            + ocaml_string + ocaml_char
3255            + ( 1 - S "{}()[];" ) ) ^ 0 ,
3256       F = ( "{" * V "F" * "}"
3257            + "(" * V "F" * ")"
3258            + "[" * V "F" * "]"
3259            + ocaml_string + ocaml_char
3260            + ( 1 - S "{}()[]\\"" ) ) ^ 0
3261   }

```

```

3262 local OneFieldDefinition =
3263   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3264   * K ( 'Name.Field' , identifier ) * SkipSpace
3265   * Q ":" * SkipSpace
3266   * K ( 'TypeExpression' , expression_for_fields_type )
3267   * SkipSpace

```

```

3268 local OneField =
3269   K ( 'Name.Field' , identifier ) * SkipSpace
3270   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3271   * ( C ( expression_for_fields_value ) / ParseAgain )
3272   * SkipSpace

```

The records.

```

3273 local RecordVal =
3274   Q "{" * SkipSpace
3275   *
3276   (
3277     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3278   ) ^ -1
3279   *
3280   (
3281     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3282   )
3283   * SkipSpace
3284   * Q ";" ^ -1
3285   * SkipSpace
3286   * Comment ^ -1
3287   * SkipSpace
3288   * Q "}"
3289 local RecordType =
3290   Q "{" * SkipSpace
3291   *
3292   (
3293     OneFieldDefinition
3294     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3295   )
3296   * SkipSpace
3297   * Q ";" ^ -1
3298   * SkipSpace
3299   * Comment ^ -1
3300   * SkipSpace
3301   * Q "}"
3302 local Record = RecordType + RecordVal

```

```

3303 local Operator =
3304   P "||" *

```

Don't use \hspace instead of \kern!

```

3305 Lc([{\PitonStyle{Operator}{\kern0.1em/\kern-0.2em/\kern0.1em}}])
3306 +
3307 K ( 'Operator' ,
3308   P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3309   "//" + "*" + ";" + "->" + "+." + "-." + "*." + "/" +
3310   + S "--+/*%=<>&@|" )

3311 local Builtin =
3312   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

```

```

3313 local Exception =
3314   K ( 'Exception' ,
3315     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3316     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3317     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

3318 LPEG_cleaner.ocaml = Compute_LPEG_cleaner ( 'ocaml' , braces )

```

An argument in the definition of a OCaml function may be of the form (pattern:type). pattern may be a single identifier but it's not mandatory. First instance, it's possible to write in OCaml:

```
let head (a::q) = a
```

First, we write a pattern (in the LPEG sens!) to match what will be the pattern (in the OCaml sens).

```

3319 local pattern_part =
3320   ( P "(" * balanced_parens * ")" + ( 1 - S ":" ) + P ":" ) ^ 0

```

For the “type” part, the LPEG-pattern will merely be `balanced_parens`.

We can now write a LPEG `Argument` which catches a argument of function (in the definition of the function).

```
3321 local Argument =
```

The following line is for the labels of the labeled arguments. Maybe we will, in the future, create a style for those elements.

```

3322   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3323   *

```

Now, the argument itself, either a single identifier, or a construction between parentheses

```

3324   (
3325     K ( 'Identifier.Internal' , identifier )
3326     +
3327     Q "(" * SkipSpace
3328     * ( C ( pattern_part ) / ParseAgain )
3329     * SkipSpace

```

Of course, the specification of type is optional.

```

3330     * ( Q ":" * #(1- P"=")
3331         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3332     ) ^ -1
3333     * Q ")"
3334   )

```

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

```

3335 local DefFunction =
3336   K ( 'Keyword.Governing' , "let open" )
3337   * Space
3338   * K ( 'Name.Module' , cap_identifier )
3339   +
3340   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3341   * Space
3342   * K ( 'Name.Function.Internal' , identifier )
3343   * Space
3344   * (

```

We use here the argument `strict` in order to allow a correct analyse of `let x = \uncover<2->{y}` (elsewhere, it's interpreted as a definition of a OCaml function).

```

3345     Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3346     +
3347     Argument * ( SkipSpace * Argument ) ^ 0
3348     * (
3349       SkipSpace
3350       * Q ":" * #( 1 - P "=" )
3351       * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3352     ) ^ -1
3353   )

```

## DefModule

```

3354 local DefModule =
3355   K ( 'Keyword.Governing' , "module" ) * Space
3356   *
3357   (
3358     K ( 'Keyword.Governing' , "type" ) * Space
3359     * K ( 'Name.Type' , cap_identifier )
3360     +
3361     K ( 'Name.Module' , cap_identifier ) * SkipSpace
3362     *
3363     (
3364       Q "(" * SkipSpace
3365       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3366       * Q ":" * # ( 1 - P "=" ) * SkipSpace
3367       * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3368       *
3369       (
3370         Q "," * SkipSpace
3371         * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3372         * Q ":" * # ( 1 - P "=" ) * SkipSpace
3373         * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3374       ) ^ 0
3375       * Q ")"
3376     ) ^ -1
3377   *
3378   (
3379     Q "=" * SkipSpace
3380     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3381     * Q "("
3382     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3383     *
3384     (
3385       Q ","
3386       *
3387       K ( 'Name.Module' , cap_identifier ) * SkipSpace
3388     ) ^ 0
3389     * Q ")"
3390   ) ^ -1
3391 )
3392 +
3393 K ( 'Keyword.Governing' , P "include" + "open" )
3394 * Space
3395 * K ( 'Name.Module' , cap_identifier )

```

## DefType

```

3396 local DefType =
3397   K ( 'Keyword.Governing' , "type" )
3398   * Space
3399   * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3400   * SkipSpace
3401   * ( Q "+=" + Q "=" )
3402   * SkipSpace
3403   * (
3404     RecordType
3405     +

```

The following lines are a suggestion of Y. Salmon.

```

3406   WithStyle
3407   (
3408     'TypeExpression' ,
3409     (
3410       (
3411         EOL

```

```

3412         + comment
3413         + Q ( 1
3414             - P ";;"
3415             - P "type"
3416             - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3417         )
3418     ) ^ 0
3419     *
3420     (
3421         # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3422         + Q ";;"
3423         + -1
3424     )
3425 )
3426 )
3427 )

3428 local prompt =
3429     Q "utop[" * digit^1 * Q "> "
3430 local start_of_line = P(function(subject, position)
3431     if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3432         return position
3433     end
3434     return nil
3435 end)
3436 local Prompt = #start_of_line * K( 'Prompt', prompt )
3437 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3438                 * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3439                 * ( K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

## The main LPEG for the language OCaml

```

3440 local Main =
3441     space ^ 0 * EOL
3442     + Space
3443     + Tab
3444     + Escape + EscapeMath
3445     + Beamer
3446     + DetectedCommands
3447     + TypeParameter
3448     + String + QuotedString + Char
3449     + Comment
3450     + Prompt + Answer

```

For the labels (maybe we will write in the future a dedicated LPEG pour those tokens).

```

3451     + Q "~" * Identifier * ( Q ":" ) ^ -1
3452     + Q ":" * # (1 - P ":" ) * SkipSpace
3453         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3454     + Exception
3455     + DefType
3456     + DefFunction
3457     + DefModule
3458     + Record
3459     + Keyword * EndKeyword
3460     + OperatorWord * EndKeyword
3461     + Builtin * EndKeyword
3462     + DotNotation * EndKeyword
3463     + Constructor
3464     + Identifier
3465     + Punct
3466     + Delim -- Delim is before Operator for a correct analysis of [| et |]
3467     + Operator

```

```

3468     + Number
3469     + Word

```

Here, we must not put `local`, of course.

```

3470     LPEG1.ocaml = Main ^ 0

```

```

3471     LPEG2.ocaml =
3472     Ct (

```

The following lines are in order to allow, in `\piton` (and not in `{Piton}`), judgments of type (such as `f : my_type -> 'a list`) or single expressions of type such as `my_type -> 'a list` (in that case, the argument of `\piton` *must* begin by a colon).

```

3473         (
3474         (
3475             P ":"
3476             +
3477             (
3478                 ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3479                 * Identifier
3480                 * SkipSpace
3481                 * Q ":"
3482             )
3483         )
3484         * # ( 1 - S ":@" )
3485         * SkipSpace
3486         * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 ) * -1
3487     )
3488     +
3489     (
3490         ( space ^ 0 * "\r" ) ^ -1
3491         * Lc [[ \@@_begin_line: ]]
3492         * LeadingSpace ^ 0
3493         * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
3494           + space ^ 0 * EOL
3495           + Main
3496         ) ^ 0
3497         * -1
3498         * Lc [[ \@@_end_line: ]]
3499     )
3500 )

```

End of the Lua scope for the language OCaml.

```

3501 end

```

### 3.6 The language C

We open a Lua local scope for the language C (of course, there will be also global definitions).

```

3502 --c C c++ C++
3503 do

```

```

3504     local Delim = Q ( S "{[()]}")
3505     local Punct = Q ( S ",:;!")

```

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

```

3506     local identifier = letter * alphanum ^ 0
3507
3508     local Operator =

```

```

3509     K ( 'Operator' ,
3510         P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3511         + S "~+/*%=<>&.@|!" )
3512
3513     local Keyword =
3514         K ( 'Keyword' ,
3515             P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3516             "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3517             "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3518             "register" + "restricted" + "return" + "static" + "static_assert" +
3519             "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3520             "union" + "using" + "virtual" + "volatile" + "while"
3521         )
3522         + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3523
3524     local Builtin =
3525         K ( 'Name.Builtin' ,
3526             P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3527
3528     local Type =
3529         K ( 'Name.Type' ,
3530             P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3531             "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3532             "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3533             "void" + "wchar_t" ) * Q "*" ^ 0
3534
3535     local DefFunction =
3536         Type
3537         * Space
3538         * Q "*" ^ -1
3539         * K ( 'Name.Function.Internal' , identifier )
3540         * SkipSpace
3541         * # P "("

```

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

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

Example: `class myclass:`

```

3542     local DefClass =
3543         K ( 'Keyword' , "class" ) * Space * K ( 'Name.Class' , identifier )

```

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

```

3544     local Character =
3545         K ( 'String.Short' ,
3546             P "[[\'\\\'\'\']]" + P "''" * ( 1 - P "''" ) ^ 0 * P "''" )

```

## The strings of C

```

3547     String =
3548         WithStyle ( 'String.Long.Internal' ,
3549             Q "\"\"
3550             * ( SpaceInString
3551                 + K ( 'String.Interpol' ,
3552                     "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3553                 )
3554             + Q ( ( P "\\\"\" + 1 - S " \" \" ) ^ 1 )
3555             ) ^ 0
3556             * Q "\"\"
3557         )

```



**Beamer** The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

3558 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
3559 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end

3560 DetectedCommands =
3561   Compute_DetectedCommands ( 'c' , braces )
3562   + Compute_RawDetectedCommands ( 'c' , braces )

3563 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

## The directives of the preprocessor

```

3564 local Preproc = K ( 'Preproc' , "#" * ( 1 - P "r" ) ^ 0 ) * ( EOL + -1 )

```

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

```

3565 local Comment =
3566   WithStyle ( 'Comment.Internal' ,
3567     Q "//" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3568     * ( EOL + -1 )
3569
3570 local LongComment =
3571   WithStyle ( 'Comment.Internal' ,
3572     Q "/*"
3573     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3574     * Q "*/"
3575     ) -- $

```

## The main LPEG for the language C

```

3576 local EndKeyword
3577   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3578     EscapeMath + -1

```

First, the main loop :

```

3579 local Main =
3580   space ^ 0 * EOL
3581   + Space
3582   + Tab
3583   + Escape + EscapeMath
3584   + CommentLaTeX
3585   + Beamer
3586   + DetectedCommands
3587   + Preproc
3588   + Comment + LongComment
3589   + Delim
3590   + Operator
3591   + Character
3592   + String
3593   + Punct
3594   + DefFunction
3595   + DefClass
3596   + Type * ( Q "*" ^ -1 + EndKeyword )
3597   + Keyword * EndKeyword
3598   + Builtin * EndKeyword
3599   + Identifier
3600   + Number
3601   + Word

```

Here, we must not put `local`, of course.

```
3602   LPEG1.c = Main ^ 0
```

We recall that each line in the C code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>7</sup>.

```
3603   LPEG2.c =
3604   Ct (
3605       ( space ^ 0 * P "\r" ) ^ -1
3606       * Lc [[ \@@_begin_line: ]]
3607       * LeadingSpace ^ 0
3608       * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3609       * -1
3610       * Lc [[ \@@_end_line: ]]
3611   )
```

End of the Lua scope for the language C.

```
3612 end
```

### 3.7 The language SQL

We open a Lua local scope for the language SQL (of course, there will be also global definitions).

```
3613 --sql SQL
3614 do

3615   local LuaKeyword
3616   function LuaKeyword ( name ) return
3617       Lc [[ {\PitonStyle{Keyword}{ }}
3618       * Q ( Cmt (
3619           C ( letter * alphanum ^ 0 ) ,
3620           function ( _ , _ , a ) return a : upper ( ) == name end
3621       )
3622   )
3623   * Lc "}}"
3624 end
```

In the identifiers, we will be able to catch those contening spaces, that is to say like "last name".

```
3625   local identifier =
3626       letter * ( alphanum + "-" ) ^ 0
3627       + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"
3628   local Operator =
3629       K ( 'Operator' , P "=" + "!=" + "<>" + ">=" + ">" + "<=" + "<" + S "*/" )
```

In SQL, the keywords are case-insensitive. That's why we have a little complication. We will catch the keywords with the identifiers and, then, distinguish the keywords with a Lua function. However, some keywords will be caught in special LPEG because we want to detect the names of the SQL tables.

The following function converts a comma-separated list in a "set", that is to say a Lua table with a fast way to test whether a string belongs to that set (eventually, the indexation of the components of the table is no longer done by integers but by the strings themselves).

```
3630   local Set
3631   function Set ( list )
3632       local set = { }
3633       for _ , l in ipairs ( list ) do set[l] = true end
3634       return set
3635   end
```

---

<sup>7</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 now use the previous function `Set` to create the “sets” `set_keywords` and `set_builtin`. That list of keywords comes from [https://sqlite.org/lang\\_keywords.html](https://sqlite.org/lang_keywords.html).

```

3636 local set_keywords = Set
3637 {
3638     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3639     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3640     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3641     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3642     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3643     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3644     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3645     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3646     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3647     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3648     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3649     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
3650     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3651     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3652     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3653     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3654     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3655     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3656     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3657     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3658 }
3659
3659 local set_builtins = Set
3660 {
3661     "AVG", "COUNT", "CHAR_LENGTH", "CONCAT", "CURDATE", "CURRENT_DATE",
3662     "DATE_FORMAT", "DAY", "LOWER", "LTRIM", "MAX", "MIN", "MONTH", "NOW",
3663     "RANK", "ROUND", "RTRIM", "SUBSTRING", "SUM", "UPPER", "YEAR"
3664 }

```

The LPEG Identifier will catch the identifiers of the fields but also the keywords and the built-in functions of SQL. It will *not* catch the names of the SQL tables.

```

3665 local Identifier =
3666   C ( identifier ) /
3667   (
3668     function ( s )
3669       if set_keywords [ s : upper ( ) ] then return

```

Remind that, in Lua, it's possible to return *several* values.

```

3670     { [[{\PitonStyle{Keyword}{}}] } ,
3671     { luatexbase.catcodetables.other , s } ,
3672     { "}" } }
3673   else
3674     if set_builtins [ s : upper ( ) ] then return
3675     { [[{\PitonStyle{Name.Builtin}{}}] } ,
3676     { luatexbase.catcodetables.other , s } ,
3677     { "}" } }
3678   else return
3679     { [[{\PitonStyle{Name.Field}{}}] } ,
3680     { luatexbase.catcodetables.other , s } ,
3681     { "}" } }
3682   end
3683 end
3684 end
3685 )

```

## The strings of SQL

```

3686 local String = K ( 'String.Long.Internal' , "'" * ( 1 - P "'" ) ^ 1 * "'" )

```

**Beamer** The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

3687 local braces = Compute_braces ( "'" * ( 1 - P "'" ) ^ 1 * "'" )
3688 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
3689
3690 DetectedCommands =
3691   Compute_DetectedCommands ( 'sql' , braces )
3692   + Compute_RawDetectedCommands ( 'sql' , braces )
3693
3694 LPEG_cleaner.sql = Compute_LPEG_cleaner ( 'sql' , braces )

```

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

```

3693 local Comment =
3694   WithStyle ( 'Comment.Internal' ,
3695     Q "--" -- syntax of SQL92
3696     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3697     * ( EOL + -1 )
3698
3699 local LongComment =
3700   WithStyle ( 'Comment.Internal' ,
3701     Q "/*"
3702     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3703     * Q "*/"
3704     ) -- $

```

**The main LPEG for the language SQL**

```

3705 local EndKeyword
3706   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3707     EscapeMath + -1
3708
3709 local TableField =
3710   K ( 'Name.Table' , identifier )
3711   * Q "."
3712   * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
3713
3714 local OneField =
3715   (
3716     Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
3717     +
3718     K ( 'Name.Table' , identifier )
3719     * Q "."
3720     * K ( 'Name.Field' , identifier )
3721     +
3722     K ( 'Name.Field' , identifier )
3723   )
3724   * (
3725     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
3726     ) ^ -1
3727   * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
3728
3729 local OneTable =
3730   K ( 'Name.Table' , identifier )
3731   * (
3732     Space
3733     * LuaKeyword "AS"
3734     * Space
3735     * K ( 'Name.Table' , identifier )
3736     ) ^ -1
3737
3738 local WeCatchTableNames =

```

```

3738     LuaKeyword "FROM"
3739     * ( Space + EOL )
3740     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
3741     + (
3742         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
3743         + LuaKeyword "TABLE"
3744     )
3745     * ( Space + EOL ) * OneTable
3746     local EndKeyword
3747     = Space + Punct + Delim + EOL + Beamer
3748     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

3749     local Main =
3750         space ^ 0 * EOL
3751         + Space
3752         + Tab
3753         + Escape + EscapeMath
3754         + CommentLaTeX
3755         + Beamer
3756         + DetectedCommands
3757         + Comment + LongComment
3758         + Delim
3759         + Operator
3760         + String
3761         + Punct
3762         + WeCatchTableNames
3763         + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
3764         + Number
3765         + Word

```

Here, we must not put local, of course.

```

3766     LPEG1.sql = Main ^ 0

```

We recall that each line in the code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>8</sup>.

```

3767     LPEG2.sql =
3768         Ct (
3769             ( space ^ 0 * "\r" ) ^ -1
3770             * Lc [[ \@@_begin_line: ]]
3771             * LeadingSpace ^ 0
3772             * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3773             * -1
3774             * Lc [[ \@@_end_line: ]]
3775         )

```

End of the Lua scope for the language SQL.

```

3776     end

```

### 3.8 The language “Minimal”

We open a Lua local scope for the language “Minimal” (of course, there will be also global definitions).

```

3777     --minimal Minimal
3778     do

```

---

<sup>8</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:`

```

3779 local Punct = Q ( S ",:;!\\\" )
3780
3781 local Comment =
3782   WithStyle ( 'Comment.Internal' ,
3783     Q "\""
3784     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3785     )
3786     * ( EOL + -1 )
3787
3788 local String =
3789   WithStyle ( 'String.Short.Internal' ,
3790     Q "\""
3791     * ( SpaceInString
3792       + Q ( ( P "[\]" ) + 1 - S " \"" ) ^ 1 )
3793     ) ^ 0
3794     * Q "\""
3795     )

```

The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

3796 local braces = Compute_braces ( P "\"" * ( P "\\\"" + 1 - P "\"" ) ^ 1 * "\"" )
3797
3798 if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
3799
3800 DetectedCommands =
3801   Compute_DetectedCommands ( 'minimal' , braces )
3802   + Compute_RawDetectedCommands ( 'minimal' , braces )
3803
3804 LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
3805
3806 local identifier = letter * alphanum ^ 0
3807
3808 local Identifier = K ( 'Identifier.Internal' , identifier )
3809
3810 local Delim = Q ( S "{[()]}" )
3811
3812 local Main =
3813   space ^ 0 * EOL
3814   + Space
3815   + Tab
3816   + Escape + EscapeMath
3817   + CommentLaTeX
3818   + Beamer
3819   + DetectedCommands
3820   + Comment
3821   + Delim
3822   + String
3823   + Punct
3824   + Identifier
3825   + Number
3826   + Word

```

Here, we must not put `local`, of course.

```

3827 LPEG1.minimal = Main ^ 0
3828
3829 LPEG2.minimal =
3830   Ct (
3831     ( space ^ 0 * "\r" ) ^ -1
3832     * Lc [ [ @@_begin_line: ] ]
3833     * LeadingSpace ^ 0
3834     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3835     * -1

```

```

3836         * Lc [[ \@@_end_line: ]]
3837     )

```

End of the Lua scope for the language “Minimal”.

```

3838 end

```

### 3.9 The language “Verbatim”

We open a Lua local scope for the language “Verbatim” (of course, there will be also global definitions).

```

3839 --verbatim Verbatim
3840 do

```

Here, we don’t use `braces` as done with the other languages because we don’t have to take into account the strings (there is no string in the language “Verbatim”).

```

3841     local braces =
3842         P { "E" ,
3843             E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3844         }
3845
3846     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
3847
3848     DetectedCommands =
3849         Compute_DetectedCommands ( 'verbatim' , braces )
3850         + Compute_RawDetectedCommands ( 'verbatim' , braces )
3851
3852     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

3853     local lpeg_central = 1 - S " \\r"
3854     if piton.begin_escape then
3855         lpeg_central = lpeg_central - piton.begin_escape
3856     end
3857     if piton.begin_escape_math then
3858         lpeg_central = lpeg_central - piton.begin_escape_math
3859     end
3860     local Word = Q ( lpeg_central ^ 1 )
3861
3862     local Main =
3863         space ^ 0 * EOL
3864         + Space
3865         + Tab
3866         + Escape + EscapeMath
3867         + Beamer
3868         + DetectedCommands
3869         + Q [[\]]
3870         + Word

```

Here, we must not put `local`, of course.

```

3871     LPEG1.verbatim = Main ^ 0
3872
3873     LPEG2.verbatim =
3874         Ct (
3875             ( space ^ 0 * "\r" ) ^ -1
3876             * Lc [[ \@@_begin_line: ]]
3877             * LeadingSpace ^ 0
3878             * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3879             * -1
3880             * Lc [[ \@@_end_line: ]]
3881         )

```

End of the Lua scope for the language “verbatim”.

```

3882 end

```

### 3.10 The language expl

We open a Lua local scope for the language `expl` of LaTeX3 (of course, there will be also global definitions).

```

3883 --EXPL expl
3884 do
3885     local Comment =
3886         WithStyle
3887         ( 'Comment.Internal' ,
3888           Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3889         )
3890     * ( EOL + -1 )

```

First, we begin with a special function to analyse the “keywords”, that is to say the control sequences beginning by “\”.

```

3891     local analyze_cs
3892     function analyze_cs ( s )
3893         local i = s : find ( ":" )
3894         if i then

```

First, the case of what might be called a “function” in `expl`, for instance, `\tl_set:Nn` or `\int_compare:nNnTF`.

```

3895         local name = s : sub ( 2 , i - 1 )
3896         local parts = name : explode ( "_" )
3897         local module = parts[1]
3898         if module == "" then module = parts[3] end

```

Remind that, in Lua, we can return *several* values.

```

3899         return
3900         { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3901           { luatexbase.catcodetables.other , s } ,
3902           { "}" } }
3903     else
3904         local p = s : sub ( 1 , 3 )
3905         if p == [[\l_]] or p == [[\g_]] or p == [[\c_]] then

```

The case of what might be called a “variable”, for instance, `\l_tmpa_int` or `\g__module_text_tl`.

```

3906         local scope = s : sub(2,2)
3907         local parts = s : explode ( "_" )
3908         local module = parts[2]
3909         if module == "" then module = parts[3] end
3910         local type = parts[#parts]
3911         return
3912         { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
3913           { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3914           { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,
3915           { luatexbase.catcodetables.other , s } ,
3916           { "}}}}}} }
3917     else

```

We have a control sequence which is neither a “function” neither a “variable” of `expl`. It’s a control sequence of standard LaTeX and we don’t format it.

```

3918         return { luatexbase.catcodetables.other , s }
3919     end
3920 end
3921 end

```

Here, we don’t use `braces` as done with the other languages because we don’t have to take into account the strings (there is no string in the language `expl`).

```

3922     local braces =
3923     P { "E" ,
3924       E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3925     }

```



```

3926
3927 if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
3928
3929 DetectedCommands =
3930   Compute_DetectedCommands ( 'expl' , braces )
3931   + Compute_RawDetectedCommands ( 'expl' , braces )
3932
3933 LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
3934 local control_sequence = P "\\\" * ( R "Az" + "_" + ":" + "@" ) ^ 1
3935 local ControlSequence = C ( control_sequence ) / analyze_cs
3936
3937 local def_function
3938   = P [[\cs_]]
3939   * ( P "set" + "new" )
3940   * ( P "_protected" ) ^ -1
3941   * P ":N" * ( P "p" ) ^ -1 * "n"
3942
3943 local DefFunction =
3944   C ( def_function ) / analyze_cs
3945   * Space
3946   * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
3947   * ControlSequence -- Q ( ControlSequence ) ?
3948   * Lc "}"
3949
3950 local Word = Q ( ( 1 - S " \r" ) ^ 1 )
3951
3952 local Main =
3953   space ^ 0 * EOL
3954   + Space
3955   + Tab
3956   + Escape + EscapeMath
3957   + Beamer
3958   + Comment
3959   + DetectedCommands
3960   + DefFunction
3961   + ControlSequence
3962   + Word

```

Here, we must not put local, of course.

```

3960 LPEG1.expl = Main ^ 0
3961
3962 LPEG2.expl =
3963   Ct (
3964     ( space ^ 0 * "\r" ) ^ -1
3965     * Lc [[ \@@_begin_line: ]]
3966     * LeadingSpace ^ 0
3967     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3968     * -1
3969     * Lc [[ \@@_end_line: ]]
3970   )

```

End of the Lua scope for the language expl of LaTeX3.

```

3971 end

```

### 3.11 The function Parse

The function `Parse` is the main function of the package `piton`. It parses its argument and sends back to LaTeX the code with interlaced formatting LaTeX instructions. In fact, everything is done by the LPEG corresponding to the considered language (`LPEG2[language]`) which returns as capture a Lua table containing data to send to LaTeX.

```

3972 function piton.Parse ( language , code )

```

The variable `piton.language` will be used by the function `ParseAgain`.

```

3973 piton.language = language
3974 local t = LPEG2[language] : match ( code )
3975 if not t then
3976     sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
3977     return -- to exit in force the function
3978 end
3979 local left_stack = {}
3980 local right_stack = {}
3981 for _ , one_item in ipairs ( t ) do
3982     if one_item == "EOL" then
3983         for i = #right_stack, 1, -1 do
3984             tex.sprint ( right_stack[i] )
3985         end

```

We remind that the `\@@_end_line:` must be explicit since it's the marker of end of the command `\@@_begin_line:`.

```

3986     sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
3987     tex.sprint ( table.concat ( left_stack ) )
3988 else

```

Here is an example of an item beginning with "Open".

```
{ "Open" , "\begin{uncoverenv}<2>" , "\end{uncoverenv}" }
```

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

```

3989     if one_item[1] == "Open" then
3990         tex.sprint ( one_item[2] )
3991         table.insert ( left_stack , one_item[2] )
3992         table.insert ( right_stack , one_item[3] )
3993     else
3994         if one_item[1] == "Close" then
3995             tex.sprint ( right_stack[#right_stack] )
3996             left_stack[#left_stack] = nil
3997             right_stack[#right_stack] = nil
3998         else
3999             tex.tprint ( one_item )
4000         end
4001     end
4002 end
4003 end
4004 end

```

There is the problem of the conventions of end of lines (`\n` in Unix and Linux but `\r\n` in Windows). The function `my_file_lines` will read a file line by line after replacement of the potential `\r\n` by `\n` (that means that we go the convention UNIX).

```

4005 local my_file_lines
4006 function my_file_lines ( filename )
4007     local f = io.open ( filename , 'rb' )
4008     local s = f : read ( '*a' )
4009     f : close ( )

```

À la fin, on doit bien mettre `(.-)` et pas `(.*)`.

```

4010     return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
4011 end

```

Recall that, in Lua, `gmatch` returns an *iterator*.

```

4012 function piton.ReadFile ( name , first_line , last_line )
4013     local s = ''
4014     local i = 0
4015     for line in my_file_lines ( name ) do

```

```

4016     i = i + 1
4017     if i >= first_line then
4018         s = s .. '\r' .. line
4019     end
4020     if i >= last_line then break end
4021 end

```

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

```

4022 if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
4023     s = s : sub ( 5 , -1 )
4024 end

4025 sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { }}] )
4026 tex.sprint ( luatexbase.catcodetables.other , s )
4027 sprintL3 ( "}" )
4028 end

4029 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
4030     local s
4031     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4032     piton.GobbleParse ( lang , n , splittable , s )
4033 end

```

### 3.12 Two variants of the function Parse with integrated preprocessors

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

```

4034 function piton.ParseBis ( lang , code )
4035     return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
4036 end

```

Of course, `gsub` spans the string only once for the substitutions, which means that `####` will be replaced by `##` as expected and not by `#`.

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

```

4037 function piton.ParseTer ( lang , code )

```

Be careful: we have to write `[[\@@_breakable_space: ]]` with a space after the name of the LaTeX command `\@@_breakable_space:`.

```

4038     return piton.Parse
4039         (
4040             lang ,
4041             code : gsub ( [[\@@_breakable_space: ]] , ' ' )
4042         )
4043 end

```

### 3.13 Preprocessors of the function Parse for gobble

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

The following LPEG returns as capture the minimal number of spaces at the beginning of the lines of code.

```

4044 local AutoGobbleLPEG =
4045     ( (
4046         P " " ^ 0 * "\r"

```

```

4047      +
4048      Ct ( C " " ^ 0 ) / table.getn
4049      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
4050    ) ^ 0
4051    * ( Ct ( C " " ^ 0 ) / table.getn
4052      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4053  ) / math.min

```

The following LPEG is similar but works with the tabulations.

```

4054 local TabsAutoGobbleLPEG =
4055   (
4056     (
4057       P "\t" ^ 0 * "\r"
4058       +
4059       Ct ( C "\t" ^ 0 ) / table.getn
4060       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
4061     ) ^ 0
4062     * ( Ct ( C "\t" ^ 0 ) / table.getn
4063       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4064   ) / math.min

```

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).

```

4065 local EnvGobbleLPEG =
4066   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
4067   * Ct ( C " " ^ 0 * -1 ) / table.getn

```

The function `gobble` gobbles  $n$  characters on the left of the code. The negative values of  $n$  have special significations.

```

4068 function piton.Gobble ( n , code )
4069   if n == 0 then return
4070     code
4071   else
4072     if n == -1 then
4073       n = AutoGobbleLPEG : match ( code )

```

for the case of an empty environment (only blank lines)

```

4074     if tonumber(n) then else n = 0 end
4075   else
4076     if n == -2 then
4077       n = EnvGobbleLPEG : match ( code )
4078     else
4079       if n == -3 then
4080         n = TabsAutoGobbleLPEG : match ( code )
4081         if tonumber(n) then else n = 0 end
4082       end
4083     end
4084   end

```

We have a second test `if n == 0` because, even if the key like `auto-gobble` is in force, it's possible that, in fact, there is no space to gobble...

```

4085     if n == 0 then return
4086       code
4087     else return

```

We will now use a LPEG that we have to compute dynamically because it depends on the value of  $n$ .

```

4088   ( Ct (
4089     ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4090     * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4091   ) ^ 0 )
4092   / table.concat

```

```

4093         ) : match ( code )
4094     end
4095 end
4096 end

```

In the following code, `n` is the value of `\l_@@_gobble_int`.  
`splittable` is the value of `\l_@@_splittable_int`.

```

4097 function piton.GobbleParse ( lang , n , splittable , code )
4098     piton.ComputeLinesStatus ( code , splittable )
4099     piton.last_code = piton.Gobble ( n , code )
4100     piton.last_language = lang

```

We count the number of lines of the computer listing. The result will be stored by Lua in `\g_@@_nb_lines_int`.

```

4101     piton.CountLines ( piton.last_code )
4102     piton.Parse ( lang , piton.last_code )
4103     piton.join_and_write ( )
4104 end

```

The following function will be used when the end user has used the key `join` or the key `write`. The value of the key `join` has been written in the Lua variable `piton.join`.

```

4105 function piton.join_and_write ( )
4106     if piton.join ~= '' then
4107         if not piton.join_files [ piton.join ] then
4108             piton.join_files [ piton.join ] = piton.get_last_code ( )
4109         else
4110             if piton.join_separation == '' then
4111                 piton.join_files [ piton.join ] =
4112                     piton.join_files [ piton.join ]
4113                     .. "\r\n"
4114                     .. piton.get_last_code ( )
4115             else
4116                 piton.join_files [ piton.join ] =
4117                     piton.join_files [ piton.join ]
4118                     .. "\r\n"
4119                     .. ( piton.join_separation : gsub ( '##' , '#' ) )
4120                     .. "\r\n"
4121                     .. piton.get_last_code ( )
4122             end
4123         end
4124     end

```

Now, if the end user has used the key `write` to write the listing of the environment on an external file (on the disk).

We have written the values of the keys `write` and `path-write` in the Lua variables `piton.write` and `piton.path_write`.

If `piton.write` is not empty, that means that the key `write` has been used for the current environment and, hence, we have to write the content of the listing on the corresponding external file.

```

4125     if piton.write ~= '' then

```

We will write on `file_name` the full name (with the path) of the file in which we will write.

```

4126         local file_name = ''
4127         if piton.path_write == '' then
4128             file_name = piton.write
4129         else

```

If `piton.path_write` is not empty, that means that we will not write on a file in the current directory but in another directory. First, we verify that that directory actually exists.

```

4130             local attr = lfs.attributes ( piton.path_write )
4131             if attr and attr.mode == "directory" then
4132                 file_name = piton.path_write .. "/" .. piton.write
4133             else

```

If the directory does *not* exist, you raise an (non-fatal) error since TeX is not able to create a new directory.

```

4134      sprintL3 [[ \@_error_or_warning:n { InexistentDirectory } ]]
4135      end
4136    end
4137    if file_name ~= '' then

```

Now, `file_name` contains the complete name of the file on which we will have to write. Maybe the file does not exist but we are sure that the directory exist.

The Lua table `piton.write_files` is a table of Lua strings corresponding to all the files that we will write on the disk in the `\AtEndDocument`. They correspond to the use of the key `write` (and `path-write`).

```

4138      if not piton.write_files [ file_name ] then
4139        piton.write_files [ file_name ] = piton.get_last_code ( )
4140      else
4141        piton.write_files [ file_name ] =
4142          piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4143      end
4144    end
4145  end
4146 end

```

The following command will be used when the end user has set `print=false`.

```

4147 function piton.GobbleParseNoPrint ( lang , n , code )
4148   piton.last_code = piton.Gobble ( n , code )
4149   piton.last_language = lang
4150   piton.join_and_write ( )
4151 end

```

The following function will be used when the key `split-on-empty-lines` is in force. With that key, the computer listing is split in chunks at the empty lines (usually between the abstract functions defined in the computer code). LaTeX will be able to change the page between the chunks. The second argument `n` corresponds to the value of the key `gobble` (number of spaces to gobble).

```

4152 function piton.GobbleSplitParse ( lang , n , splittable , code )
4153   local chunks
4154   chunks =
4155     (
4156       Ct (
4157         (
4158           P " " ^ 0 * "\r"
4159         +
4160           C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4161             - ( P " " ^ 0 * ( P "\r" + -1 ) )
4162           ) ^ 1
4163         )
4164       ) ^ 0
4165     )
4166   ) : match ( piton.Gobble ( n , code ) )
4167   sprintL3 [[ \begingroup ]]
4168   sprintL3
4169     (
4170       [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
4171       .. "language = " .. lang .. ","
4172       .. "splittable = " .. splittable .. "}"
4173     )
4174   for k , v in pairs ( chunks ) do
4175     if k > 1 then
4176       sprintL3 ( [[ \l_@_split_separation_tl ]] )
4177     end
4178     tex.print
4179     (
4180       [[\begin{}} .. piton.env_used_by_split .. "}" .. "\r"

```

```

4181         .. v
4182         .. [[\end{}} .. piton.env_used_by_split .. "}\r"
4183     )
4184 end
4185 sprintL3 [[ \endgroup ]]
4186 end

4187 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4188     local s
4189     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4190     piton.GobbleSplitParse ( lang , n , splittable , s )
4191 end

```

The following Lua string will be inserted between the chunks of code created when the key `split-on-empty-lines` is in force. It's used only once: you have given a name to that Lua string only for legibility. The token list `\l_@@_split_separation_tl` corresponds to the key `split-separation`. That token list must contain elements inserted in *vertical mode* of TeX.

```

4192 piton.string_between_chunks =
4193 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4194 .. [[ \global \g_@@_line_int = 0 ]]

```

The counter `\g_@@_line_int` will be used to control the points where the code may be broken by a change of page (see the key `splittable`).

The following public Lua function is provided to the developer.

```

4195 function piton.get_last_code ( )
4196     return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4197         : gsub ( '\r\n?' , '\n' )
4198 end

```

### 3.14 To count the number of lines

```

4199 local CountBeamerEnvironments
4200 function CountBeamerEnvironments ( code ) return
4201 (
4202     Ct (
4203         (
4204             P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4205             +
4206             ( 1 - P "\r" ) ^ 0 * "\r"
4207         ) ^ 0
4208         * ( 1 - P "\r" ) ^ 0
4209         * -1
4210     ) / table.getn
4211 ) : match ( code )
4212 end

```

The following function counts the lines of code except the lines which contains only instructions for the environments of Beamer.

It is used in `GobbleParse` and at the beginning of `\@@_composition:` (in some rare circumstances). Be careful. We have tried a version with `string.gsub` without success.

```

4213 function piton.CountLines ( code )
4214     local count
4215     count =
4216         ( Ct ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4217             *
4218             (
4219                 space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4220                 + space ^ 0
4221             ) ^ -1
4222             * -1

```

```

4223         ) / table.getn
4224     ) : match ( code )
4225     if piton.beamer then
4226         count = count - 2 * CountBeamerEnvironments ( code )
4227     end
4228     sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4229 end

```

The following function is only used once (in `piton.GobbleParse`). We have written an autonomous function only for legibility. The number of lines of the code will be stored in `\l_@@_nb_non_empty_lines_int`. It will be used to compute the largest number of lines to write (when `line-numbers` is in force).

```

4230 function piton.CountNonEmptyLines ( code )
4231     local count = 0

```

The following code is not clear. We should try to replace it by use of the `string` library of Lua.

```

4232     count =
4233         ( Ct ( ( P " " ^ 0 * "\r"
4234             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4235             * ( 1 - P "\r" ) ^ 0
4236             * -1
4237         ) / table.getn
4238     ) : match ( code )
4239     count = count + 1
4240     if piton.beamer then
4241         count = count - 2 * CountBeamerEnvironments ( code )
4242     end
4243     sprintL3
4244     ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4245 end

```

The following function stores in `\l_@@_first_line_int` and `\l_@@_last_line_int` the numbers of lines of the file `file_name` corresponding to the strings `marker_beginning` and `marker_end`. `s` is the marker of the beginning and `t` is the marker of the end.

```

4246 function piton.ComputeRange ( s , t , file_name )
4247     local first_line = -1
4248     local count = 0
4249     local last_found = false
4250     for line in io.lines ( file_name ) do
4251         if first_line == -1 then
4252             if line : sub ( 1 , #s ) == s then
4253                 first_line = count
4254             end
4255         else
4256             if line : sub ( 1 , #t ) == t then
4257                 last_found = true
4258                 break
4259             end
4260         end
4261         count = count + 1
4262     end
4263     if first_line == -1 then
4264         sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4265     else
4266         if not last_found then
4267             sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4268         end
4269     end
4270     sprintL3 (
4271         [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 '
4272         .. [[ \global \l_@@_last_line_int = ]] .. count )
4273 end

```



### 3.15 To determine the empty lines of the listings

Despite its name, the Lua function `ComputeLinesStatus` computes `piton.lines_status` but also `piton.empty_lines`.

In `piton.empty_lines`, a line will have the number 0 if it's a empty line (in fact a blank line, with only spaces) and 1 elsewhere.

In `piton.lines_status`, each line will have a status with regard the breaking points allowed (for the changes of pages).

- 0 if the line is empty and a page break is allowed;
- 1 if the line is not empty but a page break is allowed after that line;
- 2 if a page break is *not* allowed after that line (empty or not empty).

`splittable` is the value of `\l_@@_splittable_int`. However, if `splittable-on-empty-lines` is in force, `splittable` is the opposite of `\l_@@_splittable_int`.

```
4274 function piton.ComputeLinesStatus ( code , splittable )
```

The lines in the listings which correspond to the beginning or the end of an environment of Beamer (eg. `\begin{uncoverenv}`) must be retrieved (those lines have *no* number and therefore, *no* status).

```
4275     local lpeg_line_beamer
4276     if piton.beamer then
4277         lpeg_line_beamer =
4278             space ^ 0
4279             * P [[\begin{}} * beamerEnvironments * "]"
4280             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4281             +
4282             space ^ 0
4283             * P [[\end{}} * beamerEnvironments * "]"
4284     else
4285         lpeg_line_beamer = P ( false )
4286     end
4287     local lpeg_empty_lines =
4288         Ct (
4289             ( lpeg_line_beamer * "\r"
4290               +
4291               P " " ^ 0 * "\r" * Cc ( 0 )
4292               +
4293               ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4294             ) ^ 0
4295             *
4296             ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4297         )
4298         * -1
4299     local lpeg_all_lines =
4300         Ct (
4301             ( lpeg_line_beamer * "\r"
4302               +
4303               ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4304             ) ^ 0
4305             *
4306             ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4307         )
4308         * -1
```

We begin with the computation of `piton.empty_lines`. It will be used in conjunction with `line-numbers`.

```
4309     piton.empty_lines = lpeg_empty_lines : match ( code )
```

Now, we compute `piton.lines_status`. It will be used in conjunction with `splittable` and `splittable-on-empty-lines`.

Now, we will take into account the current value of `\l_@@splittable_int` (provided by the *absolute value* of the argument `splittable`).

```

4310 local lines_status
4311 local s = splittable
4312 if splittable < 0 then s = - splittable end
4313
4313 if splittable > 0 then
4314   lines_status = lpeg_all_lines : match ( code )
4315 else

```

Here, we should try to copy `piton.empty_lines` but it's not easy.

```

4316   lines_status = lpeg_empty_lines : match ( code )
4317   for i , x in ipairs ( lines_status ) do
4318     if x == 0 then
4319       for j = 1 , s - 1 do
4320         if i + j > #lines_status then break end
4321         if lines_status[i+j] == 0 then break end
4322         lines_status[i+j] = 2
4323       end
4324       for j = 1 , s - 1 do
4325         if i - j == 1 then break end
4326         if lines_status[i-j-1] == 0 then break end
4327         lines_status[i-j-1] = 2
4328       end
4329     end
4330   end
4331 end

```

In all cases (whatever is the value of `splittable-on-empty-lines`) we have to deal with both extremities of the listing to format.

First from the beginning of the code.

```

4332   for j = 1 , s - 1 do
4333     if j > #lines_status then break end
4334     if lines_status[j] == 0 then break end
4335     lines_status[j] = 2
4336   end

```

Now, from the end of the code.

```

4337   for j = 1 , s - 1 do
4338     if #lines_status - j == 0 then break end
4339     if lines_status[#lines_status - j] == 0 then break end
4340     lines_status[#lines_status - j] = 2
4341   end

```

```

4342   piton.lines_status = lines_status
4343 end

```

```

4344 function piton.TranslateBeamerEnv ( code )
4345   local s
4346   s =
4347   (
4348     Ct (
4349       (
4350         space ^ 0
4351         * C (
4352           ( P "\\begin{" + "\\end{" )
4353           * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * "\r"
4354         )
4355         + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4356       ) ^ 0
4357     *
4358     (

```

```

4359      (
4360          space ^ 0
4361          * C (
4362              ( P "\\begin{" + "\\end{" )
4363              * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * -1
4364          )
4365          + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4366      ) ^ -1
4367  )
4368  ) ^ -1 / table.concat
4369  ) : match ( code )
4370  sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4371  tex.sprint ( luatexbase.catcodetables.other , s )
4372  sprintL3 ( "]" )
4373 end

```

### 3.16 To create new languages with the syntax of listings

```

4374 function piton.new_language ( lang , definition )
4375     lang = lang : lower ( )

4376     local alpha , digit = lpeg.alpha , lpeg.digit
4377     local extra_letters = { "@" , "_" , "$" } --

```

The command `add_to_letter` (triggered by the key `)` don't write right away in the LPEG pattern of the letters in an intermediate `extra_letters` because we may have to retrieve letters from that "list" if there appear in a key alsoother.

```

4378     function add_to_letter ( c )
4379         if c ~= " " then table.insert ( extra_letters , c ) end
4380     end

```

For the digits, it's straitforward.

```

4381     function add_to_digit ( c )
4382         if c ~= " " then digit = digit + c end
4383     end

```

The main use of the key `alsoother` is, for the language LaTeX, when you have to retrieve some characters from the list of letters, in particular `@` and `_` (which, by default, are not allowed in the name of a control sequence in TeX).

(In the following LPEG we have a problem when we try to add `{` and `}`).

```

4384     local other = S "._@+~*/<>!?.() [] ~^=#&\"'\\$" --
4385     local extra_others = { }
4386     function add_to_other ( c )
4387         if c ~= " " then

```

We will use `extra_others` to retrieve further these characters from the list of the letters.

```

4388         extra_others[c] = true

```

The LPEG pattern `other` will be used in conjunction with the key `tag` (mainly for languages such as HTML and XML) for the character `/` in the closing tags `</...>`.

```

4389         other = other + P ( c )
4390     end
4391 end

```

Now, the first transformation of the definition of the language, as provided by the end user in the argument definition of `piton.new_language`.

```

4392     local def_table
4393     if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4394         def_table = {}
4395     else
4396         local strict_braces =

```

```

4397     P { "E" ,
4398         E = ( "{" * V "F" * "}" + ( 1 - S ",{" }" ) ) ^ 0 ,
4399         F = ( "{" * V "F" * "}" + ( 1 - S "{" }" ) ) ^ 0
4400     }
4401     local cut_definition =
4402     P { "E" ,
4403         E = Ct ( V "F" * ( "," * V "F" ) ^ 0 ) ,
4404         F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4405             * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4406     }
4407     def_table = cut_definition : match ( definition )
4408 end

```

The definition of the language, provided by the end user of `piton` is now in the Lua table `def_table`. We will use it *several times*.

The following LPEG will be used to extract arguments in the values of the keys (`morekeywords`, `morecomment`, `morestring`, etc.).

```

4409     local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4410     local tex_arg = tex_braced_arg + C ( 1 )
4411     local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4412     local args_for_tag
4413     = tex_option_arg
4414       * space ^ 0
4415       * tex_arg
4416       * space ^ 0
4417       * tex_arg
4418     local args_for_morekeywords
4419     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4420       * space ^ 0
4421       * tex_option_arg
4422       * space ^ 0
4423       * tex_arg
4424       * space ^ 0
4425       * ( tex_braced_arg + Cc ( nil ) )
4426     local args_for_moredelims
4427     = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4428       * args_for_morekeywords
4429     local args_for_morecomment
4430     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4431       * space ^ 0
4432       * tex_option_arg
4433       * space ^ 0
4434       * C ( P ( 1 ) ^ 0 * -1 )

```

We scan the definition of the language (i.e. the table `def_table`) in order to detect the potential key **sensitive**. Indeed, we have to catch that key before the treatment of the keywords of the language. We will also look for the potential keys `alsodigit`, `alsoletter` and `tag`.

```

4435     local sensitive = true
4436     local style_tag , left_tag , right_tag
4437     for _ , x in ipairs ( def_table ) do
4438         if x[1] == "sensitive" then
4439             if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4440                 sensitive = true
4441             else
4442                 if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4443             end
4444         end
4445         if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4446         if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4447         if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4448         if x[1] == "tag" then

```

```

4449     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4450     style_tag = style_tag or [[\PitonStyle{Tag}]]
4451 end
4452 end

```

Now, the LPEG for the numbers. Of course, it uses `digit` previously computed.

```

4453 local Number =
4454   K ( 'Number.Internal' ,
4455     ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4456       + digit ^ 0 * "." * digit ^ 1
4457       + digit ^ 1 )
4458     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4459     + digit ^ 1
4460   )
4461 local string_extra_letters = ""
4462 for _ , x in ipairs ( extra_letters ) do
4463   if not ( extra_others[x] ) then
4464     string_extra_letters = string_extra_letters .. x
4465   end
4466 end
4467 local letter = alpha + S ( string_extra_letters )
4468   + P "â" + "à" + "ç" + "ê" + "ë" + "ê" + "ï" + "î"
4469   + "ô" + "û" + "ü" + "Â" + "Ã" + "Ç" + "É" + "È" + "Ê" + "Ë"
4470   + "Ī" + "Ī" + "Ō" + "Ū" + "Ū"
4471 local alphanum = letter + digit
4472 local identifier = letter * alphanum ^ 0
4473 local Identifier = K ( 'Identifier.Internal' , identifier )

```

Now, we scan the definition of the language (i.e. the table `def_table`) for the keywords.

The following LPEG does *not* catch the optional argument between square brackets in first position.

```

4474 local split_clist =
4475   P { "E" ,
4476     E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4477       * ( P "{" ) ^ 1
4478       * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4479       * ( P "}" ) ^ 1 * space ^ 0 ,
4480     F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4481   }

```

The following function will be used if the keywords are not case-sensitive.

```

4482 local keyword_to_lpeg
4483 function keyword_to_lpeg ( name ) return
4484   Q ( Cmt (
4485     C ( identifier ) ,
4486     function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4487   end
4488   )
4489 )
4490 end
4491 local Keyword = P ( false )
4492 local PrefixedKeyword = P ( false )

```

Now, we actually treat all the keywords and also the key `moredirectives`.

```

4493 for _ , x in ipairs ( def_table )
4494 do if x[1] == "morekeywords"
4495   or x[1] == "otherkeywords"
4496   or x[1] == "moredirectives"
4497   or x[1] == "moretexcs"
4498 then
4499   local keywords = P ( false )
4500   local style = [[\PitonStyle{Keyword}]]
4501   if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
4502   style = tex_option_arg : match ( x[2] ) or style
4503   local n = tonumber ( style )

```

```

4504     if n then
4505         if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4506     end
4507     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4508         if x[1] == "moretexcs" then
4509             keywords = Q ( [[\]] .. word ) + keywords
4510         else
4511             if sensitive

```

The documentation of `lstlistings` specifies that, for the key `morekeywords`, if a keyword is a prefix of another keyword, then the prefix must appear first. However, for the `lpeg`, it's rather the contrary. That's why, here, we add the new element *on the left*.

```

4512         then keywords = Q ( word ) + keywords
4513         else keywords = keyword_to_lpeg ( word ) + keywords
4514     end
4515 end
4516 end
4517 Keyword = Keyword +
4518     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4519 end

```

Of course, the feature with the key `keywordsprefix` is designed for the languages TeX, LaTeX, et al. In that case, there is two kinds of keywords (= control sequences).

- those beginning with `\` and a sequence of characters of catcode “letter”;
- those beginning by `\` followed by one character of catcode “other”.

The following code addresses both cases. Of course, the LPEG pattern `letter` must catch only characters of catcode “letter”. That's why we have a key `alsoletter` to add new characters in that category (e.g. : when we want to format L3 code). However, the LPEG pattern is allowed to catch *more* than only the characters of catcode “other” in TeX.

```

4520     if x[1] == "keywordsprefix" then
4521         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4522         PrefixedKeyword = PrefixedKeyword
4523             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4524     end
4525 end

```

Now, we scan the definition of the language (i.e. the table `def_table`) for the strings.

```

4526     local long_string = P ( false )
4527     local Long_string = P ( false )
4528     local LongString = P ( false )
4529     local central_pattern = P ( false )
4530     for _ , x in ipairs ( def_table ) do
4531         if x[1] == "morestring" then
4532             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4533             arg2 = arg2 or [[\PitonStyle{String.Long}]]
4534             if arg1 ~= "s" then
4535                 arg4 = arg3
4536             end
4537             central_pattern = 1 - S ( " \r" .. arg4 )
4538             if arg1 : match "b" then
4539                 central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4540             end

```

In fact, the specifier `d` is point-less: when it is not in force, it's still possible to double the delimiter with a correct behaviour of `piton` since, in that case, `piton` will compose *two* contiguous strings...

```

4541         if arg1 : match "d" or arg1 == "m" then
4542             central_pattern = P ( arg3 .. arg3 ) + central_pattern
4543         end
4544         if arg1 == "m"
4545         then prefix = B ( 1 - letter - " ) - "]" )
4546         else prefix = P ( true )
4547         end

```

First, a pattern *without captures* (needed to compute braces).

```

4548     long_string = long_string +
4549         prefix
4550         * arg3
4551         * ( space + central_pattern ) ^ 0
4552         * arg4

```

Now a pattern *with captures*.

```

4553     local pattern =
4554         prefix
4555         * Q ( arg3 )
4556         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4557         * Q ( arg4 )

```

We will need Long\_string in the nested comments.

```

4558     Long_string = Long_string + pattern
4559     LongString = LongString +
4560         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4561         * pattern
4562         * Ct ( Cc "Close" )
4563     end
4564 end

```

The argument of Compute\_braces must be a pattern *which does no catching* corresponding to the strings of the language.

```

4565     local braces = Compute_braces ( long_string )
4566     if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4567
4568     DetectedCommands =
4569         Compute_DetectedCommands ( lang , braces )
4570         + Compute_RawDetectedCommands ( lang , braces )
4571
4572     LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )

```

Now, we deal with the comments and the delims.

```

4573     local CommentDelim = P ( false )
4574
4575     for _ , x in ipairs ( def_table ) do
4576         if x[1] == "morecomment" then
4577             local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4578             arg2 = arg2 or [[\PitonStyle{Comment}]]

```

If the letter i is present in the first argument (eg: morecomment = [si]{(\*){\*}), then the corresponding comments are discarded.

```

4579             if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4580             if arg1 : match "l" then
4581                 local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4582                     : match ( other_args )
4583                 if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4584                 if arg3 == [[\%]] then arg3 = "%" end -- mandatory
4585                 CommentDelim = CommentDelim +
4586                     Ct ( Cc "Open"
4587                         * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4588                         * Q ( arg3 )
4589                         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4590                         * Ct ( Cc "Close" )
4591                         * ( EOL + -1 )
4592             else
4593                 local arg3 , arg4 =
4594                     ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
4595                 if arg1 : match "s" then
4596                     CommentDelim = CommentDelim +
4597                         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4598                         * Q ( arg3 )

```

```

4599         * (
4600             CommentMath
4601             + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
4602             + EOL
4603         ) ^ 0
4604         * Q ( arg4 )
4605         * Ct ( Cc "Close" )
4606     end
4607     if arg1 : match "n" then
4608         CommentDelim = CommentDelim +
4609         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4610         * P { "A" ,
4611             A = Q ( arg3 )
4612             * ( V "A"
4613                 + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
4614                     - S "\r$" ) ^ 1 ) -- $
4615                 + long_string
4616                 + "$" -- $
4617                 * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
4618                 * "$" -- $
4619                 + EOL
4620             ) ^ 0
4621             * Q ( arg4 )
4622         }
4623         * Ct ( Cc "Close" )
4624     end
4625 end
4626 end

```

For the `moredelim`, we have to add another argument in first position, equal to `*` or `**`.

```

4627 if x[1] == "moredelim" then
4628     local arg1 , arg2 , arg3 , arg4 , arg5
4629     = args_for_moredelims : match ( x[2] )
4630     local MyFun = Q
4631     if arg1 == "*" or arg1 == "**" then
4632         function MyFun ( x )
4633             if x ~= '' then return
4634                 LPEG1[lang] : match ( x )
4635             end
4636         end
4637     end
4638     local left_delim
4639     if arg2 : match "i" then
4640         left_delim = P ( arg4 )
4641     else
4642         left_delim = Q ( arg4 )
4643     end
4644     if arg2 : match "l" then
4645         CommentDelim = CommentDelim +
4646         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4647         * left_delim
4648         * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4649         * Ct ( Cc "Close" )
4650         * ( EOL + -1 )
4651     end
4652     if arg2 : match "s" then
4653         local right_delim
4654         if arg2 : match "i" then
4655             right_delim = P ( arg5 )
4656         else
4657             right_delim = Q ( arg5 )
4658         end
4659         CommentDelim = CommentDelim +
4660         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )

```



```

4661      * left_delim
4662      * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
4663      * right_delim
4664      * Ct ( Cc "Close" )
4665    end
4666  end
4667 end
4668
4669 local Delim = Q ( S "{[()]}" )
4670 local Punct = Q ( S "=:;!\\"'" )
4671
4672 local Main =
4673   space ^ 0 * EOL
4674   + Space
4675   + Tab
4676   + Escape + EscapeMath
4677   + CommentLaTeX
4678   + Beamer
4679   + DetectedCommands
4680   + CommentDelim

```

We must put `LongString` before `Delim` because, in PostScript, the strings are delimited by parenthesis and those parenthesis would be caught by `Delim`.

```

4680   + LongString
4681   + Delim
4682   + PrefixedKeyword
4683   + Keyword * ( -1 + # ( 1 - alphanum ) )
4684   + Punct
4685   + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4686   + Number
4687   + Word

```

The LPEG `LPEG1[lang]` is used to reformat small elements, for example the arguments of the “detected commands”.

Of course, here, we must not put `local`, of course.

```

4688   LPEG1[lang] = Main ^ 0

```

The LPEG `LPEG2[lang]` is used to format general chunks of code.

```

4689   LPEG2[lang] =
4690     Ct (
4691       ( space ^ 0 * P "\r" ) ^ -1
4692       * Lc [[ \@@_begin_line: ]]
4693       * LeadingSpace ^ 0
4694       * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
4695       * -1
4696       * Lc [[ \@@_end_line: ]]
4697     )

```

If the key `tag` has been used. Of course, this feature is designed for the languages such as HTML and XML.

```

4698   if left_tag then
4699     local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
4700     * Q ( left_tag * other ^ 0 ) -- $
4701     * ( ( 1 - P ( right_tag ) ) ^ 0 )
4702     / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
4703     * Q ( right_tag )
4704     * Ct ( Cc "Close" )
4705
4706     MainWithoutTag
4707     = space ^ 1 * -1
4708     + space ^ 0 * EOL
4709     + Space
4710     + Tab
4711     + Escape + EscapeMath
4712     + CommentLaTeX
4713     + Beamer

```

```

4713         + DetectedCommands
4714         + CommentDelim
4715         + Delim
4716         + LongString
4717         + PrefixedKeyword
4718         + Keyword * ( -1 + # ( 1 - alphanum ) )
4719         + Punct
4720         + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4721         + Number
4722         + Word
4723 LPEG0[lang] = MainWithoutTag ^ 0
4724 local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
4725               + Beamer + DetectedCommands + CommentDelim + Tag
4726 MainWithTag
4727     = space ^ 1 * -1
4728     + space ^ 0 * EOL
4729     + Space
4730     + LPEGaux
4731     + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
4732 LPEG1[lang] = MainWithTag ^ 0
4733 LPEG2[lang] =
4734     Ct (
4735         ( space ^ 0 * P "\r" ) ^ -1
4736         * Lc [[ \@_begin_line: ]]
4737         * Beamer
4738         * LeadingSpace ^ 0
4739         * LPEG1[lang]
4740         * -1
4741         * Lc [[ \@_end_line: ]]
4742     )
4743 end
4744 end

```

### 3.17 We write the files (key 'write') and join the files in the PDF (key 'join')

```

4745 function piton.write_files_now ( )
4746     for file_name , file_content in pairs ( piton.write_files ) do
4747         local file = io.open ( file_name , "w" )
4748         if file then
4749             file : write ( file_content )
4750             file : close ( )
4751         else
4752             sprintL3
4753             ( [[ \@_error_or_warning:nn { FileError } { ]] .. file_name .. "]" )
4754         end
4755     end
4756 end

```

### 3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

4757 function piton.utf16 ( str )
4758     local hex = { "FEFF" } -- BOM UTF-16BE
4759     for _, codepoint in utf8.codes(str) do
4760         table.insert(hex, string.format("%04X", codepoint))
4761     end
4762     return table.concat(hex)
4763 end
4764 </LUA>

```