

The package `piton`*

F. Pantigny
fpantigny@wanadoo.fr

April 20, 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.¹

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 4.12 of `piton`, at the date of 2026/04/20.

¹Recall 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:" }

```

^aEach 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`).

^bThe 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`.

^c`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 } }

```

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

```

81 \keys_define:nn { piton }
82 {
83   footnote .bool_gset:N = \g_@@_footnote_bool ,
84   footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
85   footnote .usage:n = load ,
86   footnotehyper .usage:n = load ,
87
88   beamer .bool_gset:N = \g_@@_beamer_bool ,
89   beamer .usage:n = load ,
90
91   unknown .code:n = \@@_error:n { Unknown-key-for-package }
92 }
93 \@@_msg_new:nn { Unknown-key-for-package }
94 {
95   Unknown-key.\
96   You-have-used-the-key~'\l_keys_key_str'~when-loading-piton~
97   but-the-only-keys-available-here-are~'beamer',~'footnote'~
98   and~'footnotehyper'.~Other-keys-are-available-in~
99   \token_to_str:N \PitonOptions.\
100   That-key-will-be-ignored.
101 }

```

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

```

102 \ProcessKeyOptions

```

```

103 \IfClassLoadedT { beamer } { \bool_gset_true:N \g_@@_beamer_bool }
104 \IfClassLoadedT { ltx-talk } { \bool_gset_true:N \g_@@_beamer_bool }
105 \IfPackageLoadedT { beamerarticle } { \bool_gset_true:N \g_@@_beamer_bool }
106 \lua_now:e
107 {
108     piton = piton-or-{ }
109     piton.last_code = ''
110     piton.last_language = ''
111     piton.join = ''
112     piton.write = ''
113     piton.path_write = ''
114     \bool_if:NT \g_@@_beamer_bool { piton.beamer = true }
115 }

116 \RequirePackage { xcolor }

117 \@@_msg_new:nn { footnote-with-footnotehyper-package }
118 {
119     Footnote-forbidden.\
120     You-can't-use-the-option-'footnote'~because-the-package~
121     footnotehyper-has-already-been-loaded.~
122     If-you-want,~you-can-use-the-option-'footnotehyper'~and-the-footnotes~
123     within-the-environments-of-piton-will-be-extracted-with-the-tools~
124     of-the-package-footnotehyper.\
125     If-you-go-on,~the-package-footnote-won't-be-loaded.
126 }

127 \@@_msg_new:nn { footnotehyper-with-footnote-package }
128 {
129     You-can't-use-the-option-'footnotehyper'~because-the-package~
130     footnote-has-already-been-loaded.~
131     If-you-want,~you-can-use-the-option-'footnote'~and-the-footnotes~
132     within-the-environments-of-piton-will-be-extracted-with-the-tools~
133     of-the-package-footnote.\
134     If-you-go-on,~the-package-footnotehyper-won't-be-loaded.
135 }

136 \bool_if:NT \g_@@_footnote_bool
137 {

```

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

```

138 \IfClassLoadedTF { beamer }
139 { \bool_gset_false:N \g_@@_footnote_bool }
140 {
141     \IfPackageLoadedTF { footnotehyper }
142     { \@@_error:n { footnote-with-footnotehyper-package } }
143     { \usepackage { footnote } }
144 }
145 }

146 \bool_if:NT \g_@@_footnotehyper_bool
147 {

```

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

```

148 \IfClassLoadedTF { beamer }
149 { \bool_gset_false:N \g_@@_footnote_bool }
150 {
151     \IfPackageLoadedTF { footnote }
152     { \@@_error:n { footnotehyper-with-footnote-package } }
153     { \usepackage { footnotehyper } }
154     \bool_gset_true:N \g_@@_footnote_bool
155 }
156 }

```

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

```
157 \dim_new:N \l_@@_rounded_corners_dim
158 \bool_new:N \l_@@_in_label_bool
159 \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`).

```
160 \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:`).

```
161 \box_new:N \g_@@_output_box
```

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

```
162 \str_new:N \l_piton_language_str
163 \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.

```
164 \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`).

```
165 \seq_new:N \l_@@_path_seq
```

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

```
166 \seq_new:N \g_@@_join_seq
167 \str_new:N \l_@@_join_str
```

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.

```
168 \dim_new:N \l_@@_tcb_margins_dim
```

The following parameter corresponds to the key `box`.

```
169 \str_new:N \l_@@_box_str
```

In order to have a better control over the keys.

```
170 \bool_new:N \l_@@_in_PitonOptions_bool
171 \bool_new:N \l_@@_in_PitonInputFile_bool
```

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.

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

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

```
173 \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).

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

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

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

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

```
176 \int_new:N \l_@@_bg_colors_int  
  
177 \tl_new:N \l_@@_space_in_string_tl
```

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

```
178 \str_new:N \l_@@_begin_range_str  
179 \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).

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

The argument of `\PitonInputFile`.

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

The following line can't be deleted.

```
182 \bool_new:N \l_@@_tcolorbox_bool
```

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

```
183 \bool_new:N \l_@@_paperclip_bool  
184 \str_new:N \l_@@_paperclip_str
```

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

```
185 \int_new:N \g_@@_paperclip_int
```

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

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

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

```
187 \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).

```
188 \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.

```
189 \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 force²).

```
190 \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:`.

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

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

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

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

```
193 \dim_new:N \l_@@_left_margin_dim
194 \dim_new:N \l_@@_right_margin_dim
```

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

```
195 \bool_new:N \l_@@_left_margin_auto_bool
196 \bool_new:N \l_@@_right_margin_auto_bool
```

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.

```
197 \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.

```
198 \seq_new:N \g_@@_languages_seq
199 \cs_new_protected:Npn \@@_tab:
200 {
201   \bool_if:NTF \l_@@_show_spaces_bool
202   {
203     \hbox_set:Nn \l_tmpa_box
204     { \prg_replicate:nn \l_@@_tab_size_int { ~ } }
205     \dim_set:Nn \l_tmpa_dim { \box_wd:N \l_tmpa_box }
206     \(\mathcolor { gray }
207       { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \)
208   }
209   { \hbox:n { \prg_replicate:nn \l_@@_tab_size_int { ~ } } }
210   \int_gadd:Nn \g_@@_indentation_int \l_@@_tab_size_int
211 }
```

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

```
212 \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.

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

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

```
214 \cs_new_protected:Npn \@@_label:n #1
215 {
216   \bool_if:NTF \l_@@_line_numbers_bool
217   {
218     \@bsphack
219     \protected@write \@auxout { }
220     {
221       \string \newlabel { #1 }
222       {
223         { \int_use:N \g_@@_visual_line_int }
224         { \thepage }
225         { }
226         { line.#1 }
227         { }
228       }
229     }
230     \@esphack
231     \IfPackageLoadedT { hyperref }
```



```

232         { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
233     }
234     { \@@_error:n { label-with-lines-numbers } }
235 }

```

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`.

```

236 \cs_new_protected:Npn \@@_zlabel:n #1
237 {
238     \bool_if:NTF \l_@@_line_numbers_bool
239     {
240         \@bsphack
241         \protected@write \@auxout { }
242         {
243             \string \zref@newlabel { #1 }
244             {
245                 \string \default { \int_use:N \g_@@_visual_line_int }
246                 \string \page { \thepage }
247                 \string \zc@type { line }
248                 \string \anchor { line.#1 }
249             }
250         }
251         \@esphack
252         \IfPackageLoadedT { hyperref }
253         { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
254     }
255     { \@@_error:n { label-with-lines-numbers } }
256 }

```

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

```

257 \NewDocumentCommand { \@@_rowcolor:n } { o m }
258 {
259     \tl_gset:ce
260     { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 }_ t1 }
261     { \tl_if_novalue:nTF { #1 } { #2 } { [ #1 ] { #2 } } }
262     \bool_gset_true:N \g_@@_rowcolor_inside_bool
263 }

```

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).

```

264 \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.

```

265 \cs_new:Npn \@@_marker_beginning:n #1 { }
266 \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.

```

267 \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:`.

```

268 \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`.

```

269 \bool_new:N \g_@@_color_is_none_bool
270 \bool_new:N \g_@@_next_color_is_none_bool

271 \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 *wihtout* the backlash.

```

272 \clist_new:N \l_@@_detected_commands_clist
273 \clist_new:N \l_@@_raw_detected_commands_clist
274 \clist_new:N \l_@@_beamer_commands_clist
275 \clist_set:Nn \l_@@_beamer_commands_clist
276   { uncover , only , visible , invisible , alert , action }
277 \clist_new:N \l_@@_beamer_environments_clist
278 \clist_set:Nn \l_@@_beamer_environments_clist
279   { 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.

```

280 \hook_gput_code:nnn { begindocument } { . }
281   {
282     \newtoks \PitonDetectedCommands
283     \newtoks \PitonRawDetectedCommands
284     \newtoks \PitonBeamerCommands
285     \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.

```

286 \exp_args:NV \PitonDetectedCommands \l_@@_detected_commands_clist
287 \exp_args:NV \PitonRawDetectedCommands \l_@@_raw_detected_commands_clist
288 \exp_args:NV \PitonBeamerCommands \l_@@_beamer_commands_clist
289 \exp_args:NV \PitonBeamerEnvironments \l_@@_beamer_environments_clist
290 }

```

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.

```

291 \tl_new:N \g_@@_def_vertical_commands_tl

```

```

292 \cs_new_protected:Npn \@@_vertical_commands:n #1
293 {
294   \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 }
295   \clist_map_inline:nn { #1 }
296   {
297     \cs_set_eq:cc { @@ _ old _ ##1 : } { ##1 }
298     \cs_new_protected:cn { @@ _ new _ ##1 : n }
299     {
300       \bool_if:nTF
301       { \l_@@_tcolorbox_bool || ! \str_if_empty_p:N \l_@@_box_str }
302       {
303         \tl_gput_right:Nn \g_@@_after_line_tl
304         { \use:c { @@ _ old _ ##1 : } { #####1 } }
305       }
306       {
307         \cs_if_exist:cTF { g_@@_after_line _ \int_use:N \g_@@_line_int _ tl }
308         { \tl_gput_right:cn }
309         { \tl_gset:cn }
310         { g_@@_after_line _ \int_eval:n { \g_@@_line_int + 1 } _ tl }
311         { \use:c { @@ _ old _ ##1 : } { #####1 } }
312       }
313     }
314     \tl_gput_right:Nn \g_@@_def_vertical_commands_tl
315     { \cs_set_eq:cc { ##1 } { @@ _ new _ ##1 : n } }
316   }
317 }

```

2.4 Treatment of a line of code

```

318 \cs_new_protected:Npn \@@_replace_spaces:n #1
319 {
320   \tl_set:Nn \l_tmpa_tl { #1 }
321   \bool_if:NTF \l_@@_show_spaces_bool
322   {
323     \tl_set:Nn \l_@@_space_in_string_tl { } % U+2423
324     \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { } % U+2423
325   }
326   {

```

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.

```

327     \bool_if:NT \l_@@_break_lines_in_Piton_bool
328     {
329       \tl_if_eq:NnF \l_@@_space_in_string_tl { }
330       { \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.

```

331     \tl_replace_all:Nvn \l_tmpa_tl
332     \c_catcode_other_space_tl
333     \@@_breakable_space:

```

```

334     }
335   }
336   \l_tmpa_tl
337 }
338 \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:`).

```

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

340 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
341 {
342   \group_begin:
343   \int_gzero:N \g_@@_indentation_int

```

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

```

344   \hbox_set:Nn \l_@@_line_box
345   {
346     \skip_horizontal:N \l_@@_left_margin_dim
347     \bool_if:NT \l_@@_line_numbers_bool
348     {

```

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

```

349     \int_set:Nn \l_tmpa_int
350     {
351       \lua_now:e
352       {
353         tex.sprint
354         (

```

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`.

```

355         piton.empty_lines
356         [ \int_eval:n { \g_@@_line_int + 1 } ]
357       )
358     }
359   }
360   \bool_lazy_or:nnT
361   { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
362   { ! \l_@@_skip_empty_lines_bool }
363   { \int_gincr:N \g_@@_visual_line_int }
364
365   \bool_lazy_or:nnTF
366   { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
367   { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
368   {
369     \bool_lazy_or:nnTF
370     { \int_compare_p:nNn { \l_@@_numbers_step_int } = 1 }
371     {
372       \int_compare_p:nNn
373       {
374         \int_mod:nn
375         { \g_@@_visual_line_int }
376         { \l_@@_numbers_step_int }
377       }
378       = \c_one_int
379     }
380   {

```

```

381         \str_if_eq:eeTF \l_@@_line_numbers_position_str { left }
382         \@@_print_number_left:
383         {
384             \seq_gput_right:Ne \g_@@_visual_line_numbers_seq
385             { \int_use:N \g_@@_visual_line_int }
386         }
387     }
388     {
389         \str_if_eq:eeTF \l_@@_line_numbers_position_str { right }
390         { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
391     }
392 }
393 {
394     \str_if_eq:eeTF \l_@@_line_numbers_position_str { right }
395     { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
396 }
397 }

```

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).

```

398     \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
399     {
... but if only if the key left-margin is not used !
400         \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
401         { \skip_horizontal:n { 0.5 em } }
402     }

```

```

403     \bool_if:NTF \l_@@_minimize_width_bool
404     {
405         \hbox_set:Nn \l_tmpa_box
406         {
407             \language = -1
408             \raggedright
409             \strut
410             \@@_replace_spaces:n { #1 }
411             \strut \hfil
412         }
413         \dim_compare:nNnTF { \box_wd:N \l_tmpa_box } < \l_@@_code_width_dim
414         { \box_use:N \l_tmpa_box }
415         { \@@_vtop_of_code:n { #1 } }
416     }
417     { \@@_vtop_of_code:n { #1 } }
418 }

```

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

```

419     \box_set_dp:Nn \l_@@_line_box { \box_dp:N \l_@@_line_box + 1.25 pt }
420     \box_set_ht:Nn \l_@@_line_box { \box_ht:N \l_@@_line_box + 1.25 pt }
421     \box_use_drop:N \l_@@_line_box
422     \group_end:
423     \g_@@_after_line_tl
424     \tl_gclear:N \g_@@_after_line_tl
425 }

```

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

```

426     \cs_new_protected:Npn \@@_vtop_of_code:n #1
427     {
428         \vbox_top:n
429         {
430             \hsize = \l_@@_code_width_dim
431             \language = -1
432             \raggedright

```

```

433     \strut
434     \@@_replace_spaces:n { #1 }
435     \strut \hfil
436   }
437 }

```

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::`.

```

438 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_line_and_use:
439 {
440   \vtop
441   {
442     \offinterlineskip
443     \hbox
444     {

```

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`.

```

445       \group_begin:
446       \@@_compute_and_set_color:

```

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

```

447       \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
448       \bool_if:NT \g_@@_next_color_is_none_bool
449       { \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.

```

450       \bool_if:NTF \g_@@_color_is_none_bool
451       { \dim_zero:N \l_tmpb_dim }
452       { \dim_set_eq:NN \l_tmpb_dim \l_@@_listing_width_dim }
453       \dim_set:Nn \l_@@_tmpc_dim { \box_ht:N \l_@@_line_box }

```

Now, the colored panel.

```

454       \dim_compare:nNnTF \l_@@_rounded_corners_dim > \c_zero_dim
455       {
456         \int_compare:nNnTF \g_@@_line_int = \c_one_int
457         {
458           \begin{tikzpicture}[baseline = 0cm]
459             \fill (0,0)
460               [rounded-corners = \l_@@_rounded_corners_dim]
461               -- (0,\l_@@_tmpc_dim)
462               -- (\l_tmpb_dim,\l_@@_tmpc_dim)
463               [sharp-corners] -- (\l_tmpb_dim,-\l_tmpa_dim)
464               -- (0,-\l_tmpa_dim)
465               -- cycle ;
466           \end{tikzpicture}
467         }
468         {
469           \int_compare:nNnTF \g_@@_line_int = \g_@@_nb_lines_int
470           {
471             \begin{tikzpicture}[baseline = 0cm]
472               \fill (0,0) -- (0,\l_@@_tmpc_dim)
473                 -- (\l_tmpb_dim,\l_@@_tmpc_dim)
474                 [rounded-corners = \l_@@_rounded_corners_dim]
475                 -- (\l_tmpb_dim,-\l_tmpa_dim)
476                 -- (0,-\l_tmpa_dim)
477                 -- cycle ;
478             \end{tikzpicture}
479           }

```

```

480      {
481      \vrule height \l_@@_tmpc_dim
482      depth \l_tmpa_dim
483      width \l_tmpb_dim

```

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

```

484      % added 2026-01-02
485      \dim_compare:nNnT \l_tmpb_dim = \c_zero_dim
486      { \skip_horizontal:N \l_@@_listing_width_dim }
487    }
488  }
489  {
490  {
491    \vrule height \l_@@_tmpc_dim
492    depth \l_tmpa_dim
493    width \l_tmpb_dim

```

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

```

494      % added 2026-01-02
495      \dim_compare:nNnT \l_tmpb_dim = \c_zero_dim
496      { \skip_horizontal:N \l_@@_listing_width_dim }
497    }

```

The group is for the color of the background.

```

498    \group_end:
499    % added 2026-01-02
500    \bool_if:NT \l_@@_line_numbers_bool
501    {
502      \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
503      {
504        \seq_gpop_right:NN \g_@@_visual_line_numbers_seq \l_tmpa_tl
505        \@@_print_number_right:
506      }
507    }
508  }
509  \bool_if:NT \g_@@_next_color_is_none_bool
510  { \skip_vertical:n { 2.5 pt } }
511  \skip_vertical:n { - \box_ht_plus_dp:N \l_@@_line_box }
512  \box_use_drop:N \l_@@_line_box
513 }
514 }

```

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.

```

515 \cs_set_protected:Npn \@@_compute_and_set_color:
516 {
517   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
518   { \tl_set:Nn \l_tmpa_tl { none } }
519   {
520     \int_set:Nn \l_tmpb_int
521     { \int_mod:nn \g_@@_line_int \l_@@_bg_colors_int + 1 }
522     \tl_set:Nc \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
523   }

```

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

```

524   \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
525   {
526     \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).

```

527   \cs_undefine:c { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
528 }

```

```

529 \tl_if_eq:NnTF \l_tmpa_tl { none }
530 { \bool_gset_true:N \g_@@_color_is_none_bool }
531 {
532   \bool_gset_false:N \g_@@_color_is_none_bool
533   \@@_color:o \l_tmpa_tl
534 }

```

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).

```

535 \int_compare:nNnTF { \g_@@_line_int + 1 } = \g_@@_nb_lines_int
536 { \bool_gset_false:N \g_@@_next_color_is_none_bool }
537 {
538   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
539   { \tl_set:Nn \l_tmpa_tl { none } }
540   {
541     \int_set:Nn \l_tmpb_int
542     { \int_mod:nn { \g_@@_line_int + 1 } \l_@@_bg_colors_int + 1 }
543     \tl_set:Nn \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
544   }
545   \cs_if_exist:cT { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
546   {
547     \tl_set_eq:Nc \l_tmpa_tl
548     { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
549   }
550   \tl_if_eq:NnTF \l_tmpa_tl { none }
551   { \bool_gset_true:N \g_@@_next_color_is_none_bool }
552   { \bool_gset_false:N \g_@@_next_color_is_none_bool }
553 }
554 }

```

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} }`.

```

555 \cs_set_protected:Npn \@@_color:n #1
556 {
557   \tl_if_head_eq_meaning:nNTF { #1 } [
558     {
559       \tl_set:Nn \l_tmpa_tl { #1 }
560       \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
561       \exp_last_unbraced:No \color \l_tmpa_tl
562     }
563     { \color { #1 } }
564   }
565 \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.

```

566 \cs_new_protected:Npn \@@_par:
567 {

```

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

```

568   \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.

```

569   \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.

```
570 \kern -2.5 pt
```

Now, we control page breaks after the paragraph.

```
571 \@@_add_penalty_for_the_line:
572 }
```

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`.

```
573 \cs_set_protected:Npn \@@_breakable_space:
574 {
575   \discretionary
576     { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
577     {
578       \hbox_overlap_left:n
579         {
580           {
581             \normalfont \footnotesize \color { gray }
582             \l_@@_continuation_symbol_tl
583           }
584           \skip_horizontal:n { 0.3 em }
585           \int_compare:nNnT \l_@@_bg_colors_int > \c_zero_int
586             { \skip_horizontal:n { 0.5 em } }
587         }
588       \bool_if:NT \l_@@_indent_broken_lines_bool
589         {
590           \hbox:n
591             {
592               \prg_replicate:nn { \g_@@_indentation_int } { ~ }
593               { \color { gray } \l_@@_csoi_tl }
594             }
595         }
596     }
597   { \hbox { ~ } }
598 }
```

2.5 PitonOptions

```
599 \bool_new:N \l_@@_line_numbers_bool
600 \bool_new:N \l_@@_skip_empty_lines_bool
601 \bool_set_true:N \l_@@_skip_empty_lines_bool
602 \bool_new:N \l_@@_line_numbers_absolute_bool
603 \tl_new:N \l_@@_line_numbers_format_tl
604 \tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
605 \bool_new:N \l_@@_label_empty_lines_bool
606 \bool_set_true:N \l_@@_label_empty_lines_bool
607 \int_new:N \l_@@_number_lines_start_int
608 \str_new:N \l_@@_line_numbers_position_str
609 \str_set:Nn \l_@@_line_numbers_position_str { left }
610 \bool_new:N \l_@@_resume_bool
611 \bool_new:N \g_@@_label_as_zlabel_bool

612 \keys_define:nn { PitonOptions / marker }
613 {
614   beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
615   beginning .value_required:n = true ,
616   end .cs_set:Np = \@@_marker_end:n #1 ,
```

```

617     end .value_required:n = true ,
618     include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
619     unknown .code:n = \@@_error:n { Unknown-key-for-marker }
620 }

621 \keys_define:nn { PitonOptions / line-numbers }
622 {
623     true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
624     false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
625
626     start .code:n =
627         \bool_set_true:N \l_@@_line_numbers_bool
628         \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
629     start .value_required:n = true ,
630
631     skip-empty-lines .code:n =
632         \bool_if:NF \l_@@_in_PitonOptions_bool
633         { \bool_set_true:N \l_@@_line_numbers_bool }
634         \str_if_eq:nnTF { #1 } { false }
635         { \bool_set_false:N \l_@@_skip_empty_lines_bool }
636         { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
637
638     label-empty-lines .code:n =
639         \bool_if:NF \l_@@_in_PitonOptions_bool
640         { \bool_set_true:N \l_@@_line_numbers_bool }
641         \str_if_eq:nnTF { #1 } { false }
642         { \bool_set_false:N \l_@@_label_empty_lines_bool }
643         { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
644
645     absolute .code:n =
646         \bool_if:NTF \l_@@_in_PitonOptions_bool
647         { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
648         { \bool_set_true:N \l_@@_line_numbers_bool }
649         \bool_if:NT \l_@@_in_PitonInputFile_bool
650         {
651             \bool_set_true:N \l_@@_line_numbers_absolute_bool
652             \bool_set_false:N \l_@@_skip_empty_lines_bool
653         } ,
654     absolute .value_forbidden:n = true ,
655
656     resume .code:n =
657         \bool_set_true:N \l_@@_resume_bool
658         \bool_if:NF \l_@@_in_PitonOptions_bool
659         { \bool_set_true:N \l_@@_line_numbers_bool } ,
660     resume .value_forbidden:n = true ,
661
662     sep .dim_set:N = \l_@@_numbers_sep_dim ,
663     sep .value_required:n = true ,
664     sep .initial:n = 0.7 em ,
665
666     step .int_set:N = \l_@@_numbers_step_int ,
667     step .initial:n = 1 ,
668     step .value_required:n = true ,
669
670     position .choices:nn = { left , right }
671     { \str_set_eq:NN \l_@@_line_numbers_position_str \l_keys_choice_tl } ,
672     position .value_required:n = true ,
673
674     format .tl_set:N = \l_@@_line_numbers_format_tl ,
675     format .value_required:n = true ,
676
677     unknown .code:n =
678         \@@_unknown_key:nn

```

```

679     { PitonOptions / line-numbers }
680     { Unknown~key~for~line-numbers }
681
682 }

```

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

```

683 \keys_define:nn { PitonOptions }
684 {
685     indentations-for-Foxit .choices:nn = { true , false }
686     {
687         \tl_if_eq:VnTF \l_keys_value_tl { true }
688         { \@@_define_leading_space_Foxit: }
689         { \@@_define_leading_space_normal: }
690     } ,
691     box .choices:nn = { c , t , b , m }
692     { \str_set_eq:NN \l_@@_box_str \l_keys_choice_tl } ,
693     box .default:n = c ,
694     break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
695     break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,

```

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

```

696     detected-commands .code:n =
697         \clist_if_in:nnTF { #1 } { rowcolor }
698         {
699             \@@_error:n { rowcolor-in-detected-commands }
700             \clist_set:Nn \l_tmpa_clist { #1 }
701             \clist_remove_all:Nn \l_tmpa_clist { rowcolor }
702             \clist_put_right:No \l_@@_detected_commands_clist \l_tmpa_clist
703         }
704         { \clist_put_right:Nn \l_@@_detected_commands_clist { #1 } } ,
705     detected-commands .value_required:n = true ,
706     detected-commands .usage:n = preamble ,
707     vertical-detected-commands .code:n = \@@_vertical_commands:n { #1 } ,
708     vertical-detected-commands .value_required:n = true ,
709     vertical-detected-commands .usage:n = preamble ,
710     raw-detected-commands .code:n =
711         \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 } ,
712     raw-detected-commands .value_required:n = true ,
713     raw-detected-commands .usage:n = preamble ,
714     detected-beamer-commands .code:n =
715         \@@_error_if_not_in_beamer:
716         \clist_put_right:Nn \l_@@_beamer_commands_clist { #1 } ,
717     detected-beamer-commands .value_required:n = true ,
718     detected-beamer-commands .usage:n = preamble ,
719     detected-beamer-environments .code:n =
720         \@@_error_if_not_in_beamer:
721         \clist_put_right:Nn \l_@@_beamer_environments_clist { #1 } ,
722     detected-beamer-environments .value_required:n = true ,
723     detected-beamer-environments .usage:n = preamble ,

```

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

```

724     begin-escape .code:n =
725         \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
726     begin-escape .value_required:n = true ,
727     begin-escape .usage:n = preamble ,
728
729     end-escape .code:n =
730         \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
731     end-escape .value_required:n = true ,
732     end-escape .usage:n = preamble ,
733
734     begin-escape-math .code:n =
735         \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,

```

```

736 begin-escape-math .value_required:n = true ,
737 begin-escape-math .usage:n = preamble ,
738
739 end-escape-math .code:n =
740   \lua_now:e { \piton_end_escape_math = "\lua_escape:n{#1}" } ,
741 end-escape-math .value_required:n = true ,
742 end-escape-math .usage:n = preamble ,
743
744 comment-latex .code:n = \lua_now:n { \comment_latex = "#1" } ,
745 comment-latex .value_required:n = true ,
746 comment-latex .usage:n = preamble ,
747
748 label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
749 label-as-zlabel .usage:n = preamble ,
750
751 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
752 math-comments .usage:n = preamble ,

```

Now, general keys.

```

753 language .code:n =
754   \str_set:N \l_piton_language_str { \str_lowercase:n { #1 } } ,
755 language .value_required:n = true ,
756 path .code:n =
757   \seq_clear:N \l_@@_path_seq
758   \clist_map_inline:nn { #1 }
759   {
760     \str_set:Nn \l_tmpa_str { ##1 }
761     \seq_put_right:No \l_@@_path_seq { \l_tmpa_str }
762   } ,
763 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.

```

764 path .initial:n = . ,
765 path-write .str_set:N = \l_@@_path_write_str ,
766 path-write .value_required:n = true ,
767 font-command .tl_set:N = \l_@@_font_command_tl ,
768 font-command .initial:n = \ttfamily ,
769 font-command .value_required:n = true ,
770 font-command+ .code:n
771   = { \tl_put_right:Nn \l_@@_font_command_tl { #1 } } ,
772 font-command+ .value_required:n = true ,
773 font-command~+ .meta:n = { font-command+ = #1 } ,
774 font-command~+ .value_required:n = true ,
775 gobble .int_set:N = \l_@@_gobble_int ,
776 gobble .default:n = -1 ,
777 auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
778 auto-gobble .value_forbidden:n = true ,
779 env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
780 env-gobble .value_forbidden:n = true ,
781 tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
782 tabs-auto-gobble .value_forbidden:n = true ,
783
784 splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
785
786 split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,

```

When the key `split-on-empty-lines` is in force, the correspondint 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). That parameter must contain elements to be inserted in *vertical* mode by TeX.

```

787 split-separation .tl_set:N = \l_@@_split_separation_tl ,
788 split-separation .value_required:n = true ,
789 split-separation .initial:n = \vspace { \baselineskip } \vspace { -1.25pt } ,
790

```

```

791 split-separation+ .code:n =
792   \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
793 split-separation+ .value_required:n = true ,
794 split-separation~+ .meta:n = { split-separation+ = #1 } ,
795 add-to-split-separation .meta:n = { split-separation+ = #1 } ,
796
797 marker .code:n =
798   \bool_lazy_or:nnTF
799     \l_@@_in_PitonInputFile_bool
800     \l_@@_in_PitonOptions_bool
801     { \keys_set:nn { PitonOptions / marker } { #1 } }
802     { \@@_error:n { Invalid-key } } ,
803 marker .value_required:n = true ,
804
805 line-numbers .code:n =
806   \keys_set:nn { PitonOptions / line-numbers } { #1 } ,

```

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).

```

807 splittable .int_set:N          = \l_@@_splittable_int ,
808 splittable .default:n         = 1 ,
809 splittable .initial:n         = 100 ,
810 background-color .code:n =
811   \clist_set:Nn \l_@@_bg_color_clist { #1 }

```

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

```

812   \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
813 background-color .value_required:n = true ,

```

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`).

```

814 prompt-background-color .tl_set:N          = \l_@@_prompt_bg_color_tl ,
815 prompt-background-color .value_required:n = true ,
816 prompt-background-color .initial:n        = gray!15 ,

```

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.

```

817 print .bool_set:N = \l_@@_print_bool ,
818 print .value_required:n = true ,
819 print .initial:n = true ,
820
821 width .code:n =
822   \str_if_eq:nnTF { #1 } { min }
823   {
824     \bool_set_true:N \l_@@_minimize_width_bool
825     \dim_zero:N \l_@@_width_dim
826   }
827   {
828     \bool_set_false:N \l_@@_minimize_width_bool
829     \dim_set:Nn \l_@@_width_dim { #1 }
830   } ,
831 width .value_required:n = true ,
832
833 max-width .code:n =
834   \bool_set_true:N \l_@@_minimize_width_bool
835   \dim_set:Nn \l_@@_width_dim { #1 } ,
836 max-width .value_required:n = true ,
837
838 paperclip .code:n =
839   \bool_set_true:N \l_@@_paperclip_bool
840   \tl_if_novalue:nTF { #1 }

```

```

841     { \str_set:Nn \l_@@_paperclip_str { } }
842     { \str_set:Nn \l_@@_paperclip_str { #1 } } ,
843
844     annotation .bool_set:N = \l_@@_annotation_bool ,
845
846     write .str_set:N = \l_@@_write_str ,
847     write .value_required:n = true ,
848     no-write .code:n = \str_set_eq:NN \l_@@_write_str \c_empty_str ,
849     no-write .value_forbidden:n = true ,
850     join .code:n =
851         \str_set:Nn \l_@@_join_str { #1 }
852         \seq_if_in:NnF \g_@@_join_seq { #1 }
853         { \seq_gput_right:No \g_@@_join_seq { #1 } } ,
854     join .value_required:n = true ,
855     join-separation .str_set:N = \l_@@_join_separation_str ,
856     join-separation .value_required:n = true ,
857     no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,
858     no-join .value_forbidden:n = true ,
859
860     left-margin .code:n =
861         \str_if_eq:nnTF { #1 } { auto }
862         {
863             \dim_zero:N \l_@@_left_margin_dim
864             \bool_set_true:N \l_@@_left_margin_auto_bool
865         }
866         {
867             \dim_set:Nn \l_@@_left_margin_dim { #1 }
868             \bool_set_false:N \l_@@_left_margin_auto_bool
869         } ,
870     left-margin .value_required:n = true ,
871
872     right-margin .code:n =
873         \str_if_eq:nnTF { #1 } { auto }
874         {
875             \dim_zero:N \l_@@_right_margin_dim
876             \bool_set_true:N \l_@@_right_margin_auto_bool
877         }
878         {
879             \dim_set:Nn \l_@@_right_margin_dim { #1 }
880             \bool_set_false:N \l_@@_right_margin_auto_bool
881         } ,
882     right-margin .value_required:n = true ,
883
884     tab-size .int_set:N = \l_@@_tab_size_int ,
885     tab-size .value_required:n = true ,
886     tab-size .initial:n = 4 ,
887
888     show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
889     show-spaces .value_forbidden:n = true ,
890
891     show-spaces-in-strings .code:n =
892         \tl_set:Nn \l_@@_space_in_string_tl { } , % U+2423
893     show-spaces-in-strings .value_forbidden:n = true ,
894
895     break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
896     break-lines-in-Piton .initial:n = true ,
897
898     break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
899
900     break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
901     break-lines .value_forbidden:n = true ,
902
903     indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,

```

```

904
905 end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
906 end-of-broken-line .value_required:n = true ,
907 end-of-broken-line .initial:n = \hspace* { 0.5em } \textbackslash ,
908
909 continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
910 continuation-symbol .value_required:n = true ,
911 continuation-symbol .initial:n = + ,
912
913 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
914 continuation-symbol-on-indentation .value_required:n = true ,
915 continuation-symbol-on-indentation .initial:n = $\hookrightarrow ;$ ,
916
917 first-line .code:n = \@@_in_PitonInputFile:n
918 { \int_set:Nn \l_@@_first_line_int { #1 } } ,
919 first-line .value_required:n = true ,
920
921 last-line .code:n = \@@_in_PitonInputFile:n
922 { \int_set:Nn \l_@@_last_line_int { #1 } } ,
923 last-line .value_required:n = true ,
924
925 begin-range .code:n = \@@_in_PitonInputFile:n
926 { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
927 begin-range .value_required:n = true ,
928
929 end-range .code:n = \@@_in_PitonInputFile:n
930 { \str_set:Nn \l_@@_end_range_str { #1 } } ,
931 end-range .value_required:n = true ,
932
933 range .code:n = \@@_in_PitonInputFile:n
934 {
935   \str_set:Nn \l_@@_begin_range_str { #1 }
936   \str_set:Nn \l_@@_end_range_str { #1 }
937 } ,
938 range .value_required:n = true ,
939
940 env-used-by-split .code:n =
941   \lua_now:n { piton.env_used_by_split = '#1' } ,
942 env-used-by-split .initial:n = Piton ,
943
944 resume .meta:n = line-numbers/resume ,
945
946 unknown .code:n =
947   \@@_unknown_key:nn
948   { PitonOptions }
949   { Unknown~key~for~PitonOptions } ,
950
951 % deprecated
952 all-line-numbers .code:n =
953   \bool_set_true:N \l_@@_line_numbers_bool
954   \bool_set_false:N \l_@@_skip_empty_lines_bool ,
955
956 rounded-corners .code:n =
957   \AtBeginDocument
958   {
959     \IfPackageLoadedTF { tikz }
960     { \dim_set:Nn \l_@@_rounded_corners_dim { #1 } }
961     { \@@_err_angled_corners_without_Tikz: }
962   } ,
963   rounded-corners .default:n = 4 pt
964 }
965
966 \hook_gput_code:nnn { begindocument } { . }
967 {
968   \IfPackageLoadedTF { tcolorbox }

```

```

967 {
968   \pgfkeysifdefined { / tcb / libload / breakable }
969   {
970     \keys_define:nn { PitonOptions }
971     {
972       tcolorbox .bool_set:N = \l_@@_tcolorbox_bool ,
973     }
974   }
975   {
976     \keys_define:nn { PitonOptions }
977     { tcolorbox .code:n = \@@_error:n { library-breakable-not-loaded } }
978   }
979 }
980 {
981   \keys_define:nn { PitonOptions }
982   { tcolorbox .code:n = \@@_error:n { tcolorbox-not-loaded } }
983 }
984 }

985 \cs_new_protected:Npn \@@_err_rounded_corners_without_Tikz:
986 {
987   \@@_error:n { rounded-corners-without-Tikz }
988   \cs_gset:Npn \@@_err_rounded_corners_without_Tikz: { }
989 }

990 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
991 {
992   \bool_if:NTF \l_@@_in_PitonInputFile_bool
993   { #1 }
994   { \@@_error:n { Invalid-key } }
995 }

996 \NewDocumentCommand \PitonOptions { m }
997 {
998   \bool_set_true:N \l_@@_in_PitonOptions_bool
999   \keys_set:nn { PitonOptions } { #1 }
1000   \bool_set_false:N \l_@@_in_PitonOptions_bool
1001 }

```

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.

```

1002 \NewDocumentCommand \@@_fake_PitonOptions { }
1003 { \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.

```

1004 \int_new:N \g_@@_visual_line_int
1005 \cs_new_protected:Npn \@@_incr_visual_line:
1006 { \bool_if:NF \l_@@_skip_empty_lines_bool { \int_gincr:N \g_@@_visual_line_int } }

```

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`.

```

1007 \cs_new_protected:Npn \@@_print_number_left:
1008 {

```



```

1009 \hbox_overlap_left:n
1010 {
1011     \@@_actually_print_number:n { \int_to_arabic:n { \g_@@_visual_line_int } }
1012     \skip_horizontal:N \l_@@_numbers_sep_dim
1013 }
1014 }

```

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`.

```

1015 \cs_new_protected:Npn \@@_print_number_right:
1016 {
1017     \hbox_overlap_left:n
1018     {
1019         \@@_actually_print_number:n { \l_tmpa_tl }
1020         \int_compare:nNnT \l_@@_bg_colors_int > 0
1021         { \skip_horizontal:n { 0.1 em } }
1022     }
1023 }

```

`\@@_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:`

```

1024 \cs_new_protected:Npn \@@_actually_print_number:n #1
1025 {
1026     \group_begin:
1027     \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`.

```

1028 \l_@@_line_numbers_format_tl { #1 }
1029 \pdfextension literal { EMC }
1030 \group_end:
1031 }

```

2.7 The main commands and environments for the end user

```

1032 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
1033 {
1034     \tl_if_novalue:nTF { #3 }

```

The last argument is provided by curryfication.

```

1035     { \@@_NewPitonLanguage:nnn { #1 } { #2 } }

```

The two last arguments are provided by curryfication.

```

1036     { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
1037 }

```

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.

```

1038 \prop_new:N \g_@@_languages_prop

```

```

1039 \keys_define:nn { NewPitonLanguage }
1040 {
1041     morekeywords .code:n = ,
1042     otherkeywords .code:n = ,
1043     sensitive .code:n = ,
1044     keywordsprefix .code:n = ,
1045     moretexcs .code:n = ,
1046     morestring .code:n = ,
1047     morecomment .code:n = ,
1048     moredelim .code:n = ,
1049     moredirectives .code:n = ,

```

```

1050     tag .code:n = ,
1051     alsodigit .code:n = ,
1052     alsoletter .code:n = ,
1053     alsoother .code:n = ,
1054     unknown .code:n = \@@_error:n { Unknown~key~NewPitonLanguage }
1055 }

```

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

```

1056 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
1057 {

```

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}{...}`.

```

1058     \tl_set:Nx \l_tmpa_tl
1059     {
1060         \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
1061         \str_lowercase:n { #2 }
1062     }

```

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

```

1063     \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.

```

1064     \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).

```

1065     \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
1066 }
1067 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
1068 {
1069     \hook_gput_code:nnn { begindocument } { . }
1070     { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }
1071 }
1072 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }

```

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

```

1073 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4 #5
1074 {

```

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}{...}`

```

1075     \tl_set:Nx \l_tmpa_tl
1076     {
1077         \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
1078         \str_lowercase:n { #4 }
1079     }

```

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`.

```

1080     \prop_get:NoNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl

```

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

```

1081     { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
1082     { \@@_error:n { Language~not~defined } }
1083 }

```

```

1084 \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.

```

1085 { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
1086 \cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }

1087 \NewDocumentCommand { \piton } { }
1088 { \peek_meaning:NTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }

1089 \NewDocumentCommand { \@@_piton_standard } { m }
1090 {
1091   \group_begin:
1092   \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
1093   {

```

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.

```

1094     \bool_lazy_or:nnT
1095     \l_@@_break_lines_in_piton_bool
1096     \l_@@_break_strings_anywhere_bool
1097     { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
1098   }

```

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

```

1099   \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:

```

1100   \cs_set_eq:NN \ \ \c_backslash_str
1101   \cs_set_eq:NN \% \c_percent_str
1102   \cs_set_eq:NN \{ \c_left_brace_str
1103   \cs_set_eq:NN \} \c_right_brace_str
1104   \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.

```

1105   \cs_set_eq:cN { ~ } \space
1106   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1107   \cs_set_eq:NN \rowcolor \@@_noop_rowcolor

1108   \tl_set:Ne \l_tmpa_tl
1109   {
1110     \lua_now:e
1111     { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1112     { #1 }
1113   }
1114   \bool_if:NTF \l_@@_show_spaces_bool
1115   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1116   {
1117     \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.

```

1118     { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl \space }
1119   }

```

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

```

1120   \if_mode_math:
1121   \text { \l_@@_font_command_tl \l_tmpa_tl }
1122   \else:
1123   \l_@@_font_command_tl \l_tmpa_tl
1124   \fi:
1125   \group_end:
1126 }

```

```

1127 \NewDocumentCommand { \@@_piton_verbatim } { v }
1128 {

```

```

1129 \group_begin:
1130 \automatichyphenmode = 1
1131 \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1132 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1133 \tl_set:Ne \l_tmpa_tl
1134 {
1135   \lua_now:e
1136   { piton.Parse('\l_piton_language_str',token.scan_string()) }
1137   { #1 }
1138 }
1139 \bool_if:NT \l_@@_show_spaces_bool
1140 { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1141 \if_mode_math:
1142   \text { \l_@@_font_command_tl \l_tmpa_tl }
1143 \else:
1144   \l_@@_font_command_tl \l_tmpa_tl
1145 \fi:
1146 \group_end:
1147 }

```

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).

```

1148 \cs_new_protected:Npn \@@_piton:n #1
1149 { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }
1150
1151 \cs_new_protected:Npn \@@_piton_i:n #1
1152 {
1153   \group_begin:
1154   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
1155   \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1156   \cs_set:cpn { pitonStyle _ Prompt } { }
1157   \cs_set_eq:NN \@@_leading_space: \space
1158   \cs_set_eq:NN \@@_trailing_space: \space
1159   \tl_set:Ne \l_tmpa_tl
1160   {
1161     \lua_now:e
1162     { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1163     { #1 }
1164   }
1165   \bool_if:NT \l_@@_show_spaces_bool
1166   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1167   \@@_replace_spaces:o \l_tmpa_tl
1168   \group_end:
1169 }

```

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

```

1170 \cs_new_protected:Npn \@@_pre_composition:
1171 {
1172   \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
1173   {
1174     \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).

```

1175   \str_if_empty:NF \l_@@_box_str
1176   { \bool_set_true:N \l_@@_minimize_width_bool }
1177 }

```

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).

```

1178   \dim_set:Nn \l_@@_listing_width_dim
1179   {
1180     \bool_if:NTF \l_@@_tcolorbox_bool
1181     {
1182       \l_@@_width_dim -
1183       ( \kvtcb@left@rule
1184         + \kvtcb@lefttupper
1185         + \kvtcb@boxsep * 2
1186         + \kvtcb@righttupper
1187         + \kvtcb@right@rule )
1188     }
1189     { \l_@@_width_dim }
1190   }

1191   \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1192   \automatichyphenmode = 1
1193   \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1194   \g_@@_def_vertical_commands_tl
1195   \int_gzero:N \g_@@_line_int
1196   \int_gzero:N \g_@@_nb_lines_int
1197   \dim_zero:N \parindent
1198   \dim_zero:N \lineskip
1199   \dim_zero:N \parskip

1200
1201   % added 2026-01-02
1202   \seq_gclear:N \g_@@_visual_line_numbers_seq
1203
1204   \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

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

```

1205   \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1206   { \bool_set_true:N \l_@@_bg_bool }
1207   \bool_gset_false:N \g_@@_rowcolor_inside_bool
1208   \IfPackageLoadedTF { zref-base }
1209   {
1210     \bool_if:NTF \g_@@_label_as_zlabel_bool
1211     { \cs_set_eq:NN \label \@@_zlabel:n }
1212     { \cs_set_eq:NN \label \@@_label:n }
1213     \cs_set_eq:NN \zlabel \@@_zlabel:n
1214   }
1215   { \cs_set_eq:NN \label \@@_label:n }
1216   \l_@@_font_command_tl
1217   }

```

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`.

```

1218   \cs_new_protected:Npn \@@_compute_margin:N #1
1219   {
1220     \use:e
1221     {
1222       \bool_if:NTF \l_@@_skip_empty_lines_bool
1223       { \lua_now:n { piton.CountNonEmptyLines(token.scan_argument()) } }
1224       { \lua_now:n { piton.CountLines(token.scan_argument()) } }
1225       { \l_@@_listing_tl }
1226     }
1227     \hbox_set:Nn \l_tmpa_box

```

```

1228 {
1229   \l_@@_line_numbers_format_tl
1230   \int_to_arabic:n
1231   {
1232     \g_@@_visual_line_int
1233     +
1234     \bool_if:NTF \l_@@_skip_empty_lines_bool
1235     { \l_@@_nb_non_empty_lines_int }
1236     { \g_@@_nb_lines_int }
1237   }
1238 }
1239 \dim_set:Nn #1 { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1240 }

```

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.

```

1241 \cs_new_protected:Npn \@@_compute_code_width:
1242 {
1243   \dim_set:Nn \l_@@_code_width_dim
1244   {
1245     \l_@@_listing_width_dim
1246     -
1247     (
1248       \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1249       {
1250         \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1251         { \l_@@_left_margin_dim }
1252         { 0.5 em }
1253       }
1254       +
1255       \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1256       { \l_@@_right_margin_dim }
1257       { 0.5 em }
1258     )
1259   }
1260 }
1261 }

```

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:`.

```

1262 \cs_new_protected:Npn \@@_recompute_listing_width:
1263 {
1264   \dim_set:Nn \l_@@_listing_width_dim
1265   {
1266     \box_wd:N \g_@@_output_box
1267     +
1268     \int_compare:nNnTF \l_@@_bg_colors_int > \c_zero_int
1269     {
1270       \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1271       { \l_@@_left_margin_dim }
1272       { 0.5 em }
1273     }
1274     +
1275     \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1276     { \l_@@_right_margin_dim }
1277   }

```

```

1276         { 0.5 em }
1277     }
1278     { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1279 }
1280 }

```

```

1281 \cs_new_protected:Npn \@@_store_body:n #1
1282 {

```

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

```

1283     \tl_set:Nc \obeyedline { \char_generate:nn { 13 } { 11 } }
1284     \tl_set:Nc \l_@@_listing_tl { #1 }
1285     \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1286 }

```

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

```

1287 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nnnnn
1288 {
1289     \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1290     {
1291         \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1292         #4
1293         \@@_pre_composition:
1294         \int_compare:nNt { \l_@@_number_lines_start_int } > { \c_zero_int }
1295         {
1296             \int_gset:Nn \g_@@_visual_line_int
1297                 { \l_@@_number_lines_start_int - 1 }
1298         }
1299         \bool_if:NT \g_@@_beamer_bool
1300             { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1301         \bool_if:NT \g_@@_footnote_bool \savenotes
1302         \@@_composition:
1303         \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1304             { \@@_create_paperclip_annotation: }
1305         \bool_if:NT \g_@@_footnote_bool \endsavenotes
1306         #5
1307     }
1308     { \ignorespacesafterend }
1309 }

```

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

```

1310 \cs_new_protected:Npn \@@_create_paperclip_annotation:
1311 {
1312     \marginalia
1313     {
1314         \vspace* { - 0.8 em }
1315         \hbox:n
1316         {
1317             \vrule-height~0~pt~depth~12~pt~width~0~pt
1318             \bool_if:NT \l_@@_annotation_bool
1319                 {
1320                     \lua_now:n
1321                     {

```

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!

```

1322                 pdf.immediateobj
1323                 ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1324             }
1325             \pdfextension annot~width~5pt~height~10pt~depth~0pt
1326             {
1327                 /Subtype /Text

```

```

1328         /Contents~\pdf_object_ref_last:
1329         /Name /Note
1330         /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).

```

1331         /ReplyType /Group

```

Adds the bit 10 which means LockedContents.

```

1332         /F~512
1333         /C [0.8~0.8~0.8]
1334     }
1335     \hspace* { 7 mm }
1336 }
1337 \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1338 }
1339 }
1340 }

```

```

1341 \cs_new_protected:Npn \@@_create_paperclip:
1342 {
1343     \str_if_empty:NT \l_@@_paperclip_str
1344     {
1345         \int_gincr:N \g_@@_paperclip_int
1346         \str_set:N \l_@@_paperclip_str { listing~\int_use:N \g_@@_paperclip_int .txt }
1347     }

```

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

```

1348     \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1349     \box_move_down:nn
1350     { 10 pt }
1351     {
1352         \hbox:n
1353         {
1354             \pdfextension annot~width~10pt~height~20pt~depth~0pt
1355             {
1356                 /Subtype /FileAttachment
1357                 /Name /Paperclip
1358                 /F~8 % no zoom

```

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

```

1359         /Contents (The~computer~listing)
1360         /FS <<
1361             /Type /Filespec
1362             /F (\l_@@_paperclip_str)
1363             /EF << /F~\pdf_object_ref_last: >>
1364             /AFRelationship /Supplement
1365         >>
1366     }
1367 }
1368 }
1369 }

```

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

```

1370 \NewDocumentCommand { \NewPitonEnvironment } { }
1371 { \@@_DefinePitonEnvironment:nnnnn { New } }
1372 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1373 { \@@_DefinePitonEnvironment:nnnnn { Declare } }
1374 \NewDocumentCommand { \RenewPitonEnvironment } { }
1375 { \@@_DefinePitonEnvironment:nnnnn { Renew } }
1376 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1377 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

```

```

1378 \cs_new_protected:Npn \@@_translate_beamer_env:n

```



```

1379 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1380 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1381 \cs_new_protected:Npn \@@_composition:
1382 {
1383   \str_if_empty:NT \l_@@_box_str
1384   {
1385     \mode_if_vertical:F
1386     { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1387   }

1388   \bool_if:NT \l_@@_line_numbers_bool
1389   {
1390     \bool_lazy_and:nnT
1391     { \l_@@_left_margin_auto_bool }
1392     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { left } }
1393     { \@@_compute_margin:N \l_@@_left_margin_dim }
1394     \bool_lazy_and:nnT
1395     { \l_@@_right_margin_auto_bool }
1396     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { right } }
1397     { \@@_compute_margin:N \l_@@_right_margin_dim }
1398   }

1399   \lua_now:e
1400   {
1401     piton.join_separation = "\l_@@_join_separation_str"
1402     piton.join = "\l_@@_join_str"
1403     piton.write = "\l_@@_write_str"
1404     piton.path_write = "\l_@@_path_write_str"
1405   }
1406   \noindent
1407   \bool_if:NTF \l_@@_print_bool
1408   {

```

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.

```

1409   \bool_if:NTF \l_@@_split_on_empty_lines_bool
1410   { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1411   {
1412     \@@_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).

```

1413   \bool_if:NTF \l_@@_tcolorbox_bool
1414   {
1415     \str_if_empty:NTF \l_@@_box_str
1416     \@@_composition_iii:
1417     \@@_composition_iv:
1418   }
1419   {
1420     \str_if_empty:NTF \l_@@_box_str
1421     \@@_composition_i:
1422     \@@_composition_ii:
1423   }
1424 }
1425 }
1426 { \@@_gobble_parse_no_print:o \l_@@_listing_tl }
1427 }

```

`\@@_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=...`

```
1428 \cs_new_protected:Npn \@@_composition_i:
1429 {
```

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.

```
1430 \box_clear:N \g_tmpa_box
```

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

```
1431 \box_clear_new:N \g_@@_line_box
```

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

```
1432 \vbox_set:Nn \l_tmpa_box
1433 {
1434   \vbox_unpack_drop:N \g_@@_output_box
1435   \bool_gset_false:N \g_tmpa_bool
1436   \unskip \unskip
1437   \bool_gset_false:N \g_tmpa_bool
1438   \bool_do_until:nn \g_tmpa_bool
1439   {
1440     \unskip \unskip \unskip
1441     \unpenalty \unkern
1442     \box_set_to_last:N \l_@@_line_box
1443     \box_if_empty:NTF \l_@@_line_box
1444       { \bool_gset_true:N \g_tmpa_bool }
1445     {
1446       \vbox_gset:Nn \g_tmpa_box
1447       {
1448         \vbox_unpack:N \g_tmpa_box
1449         \box_use:N \l_@@_line_box
1450       }
1451     }
1452   }
1453 }
```

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

```
1454 \bool_gset_false:N \g_tmpa_bool
1455 \int_zero:N \g_@@_line_int
1456 \bool_do_until:nn \g_tmpa_bool
1457 {
```

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`.

```
1458 \vbox_gset:Nn \g_tmpa_box
1459 {
1460   \vbox_unpack_drop:N \g_tmpa_box
1461   \box_gset_to_last:N \g_@@_line_box
1462 }
```

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.

```
1463 \box_if_empty:NTF \g_@@_line_box
1464   { \bool_gset_true:N \g_tmpa_bool }
1465   {
1466     \box_use:N \g_@@_line_box
1467     \int_gincr:N \g_@@_line_int
1468     \par
1469     \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`.

```
1470 \@@_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.

```
1471 \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 }
1472   { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 } }
```

```

1473         \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1474         { \mode_leave_vertical: }
1475     }
1476 }
1477 \skip_vertical:n { 2.5 pt }
1478 }

```

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

```

1479 \cs_new_protected:Npn \@@_composition_ii:
1480 {
1481     \use:e { \begin { minipage } [ \l_@@_box_str ] }
1482     { \l_@@_listing_width_dim }

```

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

```

1483     \vbox_unpack:N \g_@@_output_box
\kern is mandatory here (\skip_vertical:n won't work).
1484     \kern 2.5 pt
1485     \end { minipage }
1486 }

```

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

```

1487 \cs_new_protected:Npn \@@_composition_iii:
1488 {
1489     \use:e
1490     {
1491         \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.

```

1492         [ breakable , text~width = \l_@@_listing_width_dim ]
1493     }
1494     \par
1495     \vbox_unpack:N \g_@@_output_box
1496     \end { tcolorbox }
1497 }

```

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

```

1498 \cs_new_protected:Npn \@@_composition_iv:
1499 {
1500     \use:e
1501     {
1502         \begin { tcolorbox }
1503         [
1504             hbox ,
1505             text~width = \l_@@_listing_width_dim ,
1506             nobeforeafter ,
1507             box~align =
1508                 \str_case:Nn \l_@@_box_str
1509                 {
1510                     t { top }
1511                     b { bottom }
1512                     c { center }
1513                     m { center }
1514                 }
1515             ]
1516         }
1517         \box_use:N \g_@@_output_box
1518         \end { tcolorbox }
1519     }

```

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.

```

1520 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1521 {
1522   \int_case:nn
1523   {
1524     \lua_now:e
1525     {
1526       tex.sprint
1527       ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1528     }
1529   }
1530   { 1 { \penalty 100 } 2 \nobreak }
1531 }

```

`\@@_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`.

```

1532 \cs_new_protected:Npn \@@_create_output_box:
1533 {
1534   \@@_compute_code_width:
1535   \vbox_gset:Nn \g_@@_output_box
1536   { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1537   \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1538   \bool_lazy_any:nT
1539   {
1540     { \int_compare_p:nNn \l_@@_bg_colors_int > \c_zero_int }
1541     { \g_@@_rowcolor_inside_bool }
1542     {
1543       \l_@@_line_numbers_bool
1544       &&
1545       \str_if_eq_p:ee \l_@@_line_numbers_position_str { right }
1546     }
1547   }
1548   \@@_add_bg_and_right_nb_to_output_box:
1549 }

```

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:`.

```

1550 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_output_box:
1551 {
1552   \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`.

```

1553   \vbox_set:Nn \l_tmpa_box
1554   {
1555     \vbox_unpack_drop:N \g_@@_output_box

```

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

```

1556     \bool_gset_false:N \g_tmpa_bool
1557     \unskip \unskip

```

We begin the loop.

```

1558     \bool_do_until:nn \g_tmpa_bool
1559     {
1560       \unskip \unskip \unskip
1561       \int_set_eq:NN \l_tmpa_int \lastpenalty
1562       \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...

```

1563         \box_set_to_last:N \l_@@_line_box
1564         \box_if_empty:NTF \l_@@_line_box
1565         { \bool_gset_true:N \g_tmpa_bool }
1566         {
\g_@@_line_int will be used in \@@_add_bg_and_right_nb_to_line_and_use:.
1567         \vbox_gset:Nn \g_@@_output_box
1568         {

```

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`.

```

1569         \@@_add_bg_and_right_nb_to_line_and_use:
1570         \kern -2.5 pt
1571         \penalty \l_tmpa_int
1572         \vbox_unpack:N \g_@@_output_box
1573     }
1574 }
1575 \int_gdecr:N \g_@@_line_int
1576 }
1577 }
1578 }

```

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

```

1579 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1580 {
1581     \lua_now:e
1582     {
1583         piton.GobbleParseNoPrint
1584         (
1585             '\l_piton_language_str' ,
1586             \int_use:N \l_@@_gobble_int ,
1587             token.scan_argument ( )
1588         )
1589     }
1590 }
1591 \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.

```

1592 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1593 {
1594     \lua_now:e
1595     {
1596         piton.RetrieveGobbleParse
1597         (
1598             '\l_piton_language_str' ,
1599             \int_use:N \l_@@_gobble_int ,
1600             \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1601             { \int_eval:n { - \l_@@_splittable_int } }
1602             { \int_use:N \l_@@_splittable_int } ,
1603             token.scan_argument ( )
1604         )
1605     }
1606 }
1607 \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.

```

1608 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1609 {

```

```

1610 \lua_now:e
1611 {
1612     piton.RetrieveGobbleSplitParse
1613     (
1614         '\l_piton_language_str' ,
1615         \int_use:N \l_@@_gobble_int ,
1616         \int_use:N \l_@@_splittable_int ,
1617         token.scan_argument ( )
1618     )
1619 }
1620 }
1621 \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`.

```

1622 \bool_if:NTF \g_@@_beamer_bool
1623 {
1624     \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1625     {
1626         \keys_set:nn { PitonOptions } { #2 }
1627         \begin { actionenv } < #1 >
1628         }
1629         { \end { actionenv } }
1630     }
1631     {
1632         \NewPitonEnvironment { Piton } { 0 { } }
1633         { \keys_set:nn { PitonOptions } { #1 } }
1634         { }
1635     }
1636 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1637 {
1638     \mode_if_vertical:F { \par }
1639     \group_begin:
1640     \seq_concat:NNN
1641     \l_file_search_path_seq
1642     \l_@@_path_seq
1643     \l_file_search_path_seq
1644     \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1645     {
1646         \@@_input_file:nn { #1 } { #2 }
1647         #4
1648     }
1649     { #5 }
1650     \group_end:
1651 }
1652 \cs_new_protected:Npn \@@_unknown_file:n #1
1653 { \msg_error:nnn { piton } { Unknown~file } { #1 } }
1654 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1655 {
1656     \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1657     {
1658         \iow_log:n { No~file~#3 }
1659         \@@_unknown_file:n { #3 }
1660     }
1661 }
1662 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1663 {
1664     \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1665     {

```

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

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

```

1666     \iow_log:n { No-file-#3 }
1667     \@@_unknown_file:n { #3 }
1668   }
1669 }
1670 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1671 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```

1672 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1673 {

```

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 `>`).

```

1674   \tl_if_novalue:nF { #1 }
1675   {
1676     \bool_if:NTF \g_@@_beamer_bool
1677     { \begin { uncoverenv } < #1 > }
1678     { \@@_error_or_warning:n { overlay~without~beamer } }
1679   }
1680   \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.

```

1681   \iow_log:e { (\l_@@_file_name_str) }
1682   \int_zero_new:N \l_@@_first_line_int
1683   \int_zero_new:N \l_@@_last_line_int
1684   \int_set_eq:NN \l_@@_last_line_int \c_max_int
1685   \bool_set_true:N \l_@@_in_PitonInputFile_bool
1686   \keys_set:nn { PitonOptions } { #2 }
1687   \bool_if:NT \l_@@_line_numbers_absolute_bool
1688   { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1689   \bool_if:nTF
1690   {
1691     (
1692       \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1693       || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1694     )
1695     && ! \str_if_empty_p:N \l_@@_begin_range_str
1696   }
1697   {
1698     \@@_error_or_warning:n { bad-range-specification }
1699     \int_zero:N \l_@@_first_line_int
1700     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1701   }
1702   {
1703     \str_if_empty:NF \l_@@_begin_range_str
1704     {
1705       \@@_compute_range:
1706       \bool_lazy_or:nnT
1707       \l_@@_marker_include_lines_bool
1708       { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1709       {
1710         \int_decr:N \l_@@_first_line_int
1711         \int_incr:N \l_@@_last_line_int
1712       }
1713     }
1714   }
1715   \@@_pre_composition:
1716   \bool_if:NT \l_@@_line_numbers_absolute_bool
1717   { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1718   \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1719   {
1720     \int_gset:Nn \g_@@_visual_line_int
1721     { \l_@@_number_lines_start_int - 1 }

```

```
1722     }
```

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

```
1723     \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1724     { \int_gzero:N \g_@@_visual_line_int }
1725     \lua_now:e
1726     {
```

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

```
1727         piton.ReadFile(
1728             '\l_@@_file_name_str' ,
1729             \int_use:N \l_@@_first_line_int ,
1730             \int_use:N \l_@@_last_line_int )
1731     }
1732     \@@_composition:
1733     \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.

```
1734     \tl_if_novalue:nF { #1 }
1735     { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1736 }
```

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

```
1737 \cs_new_protected:Npn \@@_compute_range:
1738 {
```

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

```
1739     \str_set:Ne \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1740     \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`.

```
1741     \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1742     \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1743     \lua_now:e
1744     {
1745         piton.ComputeRange
1746         ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1747     }
1748 }
```

2.8 The styles

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

```
1749 \NewDocumentCommand { \PitonStyle } { m }
1750 {
1751     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1752     { \use:c { pitonStyle _ #1 } }
1753 }
```

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`”.

```
1754 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1755 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }
```



```

1756 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1757 {
1758   \str_clear_new:N \l_@@_SetPitonStyle_option_str
1759   \str_set:Nx \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1760   \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1761   { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1762   \keys_set:nn { piton / Styles } { #2 }
1763 }

1764 \cs_new_protected:Npn \@@_math_scantokens:n #1
1765 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

1766 \clist_new:N \g_@@_styles_clist
1767 \clist_gset:Nn \g_@@_styles_clist
1768 {
1769   Comment ,
1770   Comment.Internal ,
1771   Comment.LaTeX ,
1772   Discard ,
1773   Exception ,
1774   FormattingType ,
1775   Identifier.Internal ,
1776   Identifier ,
1777   InitialValues ,
1778   Interpol.Inside ,
1779   Keyword ,
1780   Keyword.Governing ,
1781   Keyword.Constant ,
1782   Keyword2 ,
1783   Keyword3 ,
1784   Keyword4 ,
1785   Keyword5 ,
1786   Keyword6 ,
1787   Keyword7 ,
1788   Keyword8 ,
1789   Keyword9 ,
1790   Name.Builtin ,
1791   Name.Class ,
1792   Name.Constructor ,
1793   Name.Decorator ,
1794   Name.Field ,
1795   Name.Function ,
1796   Name.Module ,
1797   Name.Namespace ,
1798   Name.Table ,
1799   Name.Type ,
1800   Number ,
1801   Number.Internal ,
1802   Operator ,
1803   Operator.Word ,
1804   Preproc ,
1805   Prompt ,
1806   String.Doc ,
1807   String.Doc.Internal ,
1808   String.Interpol ,
1809   String.Long ,
1810   String.Long.Internal ,
1811   String.Short ,
1812   String.Short.Internal ,
1813   Tag ,
1814   TypeParameter ,
1815   UserFunction ,

```

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

```
1816   TypeExpression ,
```

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

```
1817   Directive
```

```
1818 }

1819 \clist_map_inline:Nn \g_@@_styles_clist
1820 {
1821   \keys_define:nn { piton / Styles }
1822   {
1823     #1 .value_required:n = true ,
1824     #1 .code:n =
1825       \tl_set:cn
1826       {
1827         pitonStyle _
1828         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1829         { \l_@@_SetPitonStyle_option_str _ }
1830         #1
1831       }
1832     { ##1 }
1833   }
1834 }
1835
1836 \keys_define:nn { piton / Styles }
1837 {
1838   String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1839   String      .value_required:n = true ,
1840   Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1841   Comment.Math .value_required:n = true ,
1842   unknown     .code:n = \@@_unknown_style:
1843 }
```

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.

```
1844 \cs_new_protected:Npn \@@_unknown_style:
1845 {
1846   \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1847   {
1848     \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1849     \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`.

```
1850     \bool_lazy_and:nnTF
1851     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1852     {
1853       \str_if_eq_p:Vn \l_tmpa_str { Module }
1854       ||
1855       \str_if_eq_p:Vn \l_tmpa_str { Type }
1856     }
```

Now, we will create a new style.

```
1857     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1858     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1859   }
1860   { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1861 }
```

```
1862 \SetPitonStyle[OCaml]
```

```
1863 {
1864   TypeExpression =
1865   {
1866     \SetPitonStyle [ OCaml ]
```

```

1867     {
1868         Identifier = \PitonStyle { Name.Type } ,
1869         Name.Builtin = \PitonStyle { Name.Type}
1870     }
1871     \@@_piton:n
1872 }
1873 }

```

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`.

```

1874 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that clist.

```

1875 \clist_gsort:Nn \g_@@_styles_clist
1876 {
1877     \str_compare:nNnTF { #1 } < { #2 }
1878         \sort_return_same:
1879         \sort_return_swapped:
1880 }

```

```

1881 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1882
1883 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1884
1885 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1886 {
1887     \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`.

```

1888     \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space
1889     \seq_clear:N \l_tmpa_seq
1890     \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1891     \seq_use:Nn \l_tmpa_seq { \- }
1892 }

```

```

1893 \cs_new_protected:Npn \@@_comment:n #1
1894 {
1895     \PitonStyle { Comment }
1896     {
1897         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1898             {
1899                 \tl_set:Nn \l_tmpa_tl { #1 }
1900                 \tl_replace_all:NVn \l_tmpa_tl
1901                     \c_catcode_other_space_tl
1902                     \@@_breakable_space:
1903                 \l_tmpa_tl
1904             }
1905             { #1 }
1906     }
1907 }

```

```

1908 \cs_new_protected:Npn \@@_string_long:n #1
1909 {
1910     \PitonStyle { String.Long }
1911     {
1912         \bool_if:NTF \l_@@_break_strings_anywhere_bool
1913             { \@@_actually_break_anywhere:n { #1 } }
1914     }

```

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.

```

1915         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1916         {
1917             \tl_set:Nn \l_tmpa_tl { #1 }
1918             \tl_replace_all:Nvn \l_tmpa_tl
1919                 \c_catcode_other_space_tl
1920                 \@@_breakable_space:
1921             \l_tmpa_tl
1922         }
1923         { #1 }
1924     }
1925 }
1926 }
1927 \cs_new_protected:Npn \@@_string_short:n #1
1928 {
1929     \PitonStyle { String.Short }
1930     {
1931         \bool_if:NT \l_@@_break_strings_anywhere_bool
1932         { \@@_actually_break_anywhere:n }
1933         { #1 }
1934     }
1935 }
1936 \cs_new_protected:Npn \@@_string_doc:n #1
1937 {
1938     \PitonStyle { String.Doc }
1939     {
1940         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1941         {
1942             \tl_set:Nn \l_tmpa_tl { #1 }
1943             \tl_replace_all:Nvn \l_tmpa_tl
1944                 \c_catcode_other_space_tl
1945                 \@@_breakable_space:
1946             \l_tmpa_tl
1947         }
1948         { #1 }
1949     }
1950 }
1951 \cs_new_protected:Npn \@@_number:n #1
1952 {
1953     \PitonStyle { Number }
1954     {
1955         \bool_if:NT \l_@@_break_numbers_anywhere_bool
1956         { \@@_actually_break_anywhere:n }
1957         { #1 }
1958     }
1959 }

```

2.9 The initial styles

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

```

1960 \SetPitonStyle
1961 {
1962     Comment           = \color [ HTML ] { 0099FF } \itshape ,
1963     Comment.Internal  = \@@_comment:n ,
1964     Exception         = \color [ HTML ] { CC0000 } ,
1965     Keyword           = \color [ HTML ] { 006699 } \bfseries ,
1966     Keyword.Governing = \color [ HTML ] { 006699 } \bfseries ,

```

```

1967 Keyword.Constant      = \color [ HTML ] { 006699 } \bfseries ,
1968 Name.Builtin           = \color [ HTML ] { 336666 } ,
1969 Name.Decorator          = \color [ HTML ] { 9999FF } ,
1970 Name.Class              = \color [ HTML ] { 00AA88 } \bfseries ,
1971 Name.Function           = \color [ HTML ] { CC00FF } ,
1972 Name.Namespace          = \color [ HTML ] { 00CCFF } ,
1973 Name.Constructor        = \color [ HTML ] { 006000 } \bfseries ,
1974 Name.Field              = \color [ HTML ] { AA6600 } ,
1975 Name.Module             = \color [ HTML ] { 0060A0 } \bfseries ,
1976 Name.Table              = \color [ HTML ] { 309030 } ,
1977 Number                  = \color [ HTML ] { FF6600 } ,
1978 Number.Internal         = \@@_number:n ,
1979 Operator                 = \color [ HTML ] { 555555 } ,
1980 Operator.Word           = \bfseries ,
1981 String                  = \color [ HTML ] { CC3300 } ,
1982 String.Long.Internal    = \@@_string_long:n ,
1983 String.Short.Internal   = \@@_string_short:n ,
1984 String.Doc.Internal     = \@@_string_doc:n ,
1985 String.Doc              = \color [ HTML ] { CC3300 } \itshape ,
1986 String.Interpol         = \color [ HTML ] { AA0000 } ,
1987 Comment.LaTeX           = \normalfont \color [ rgb ] { .468, .532, .6 } ,
1988 Name.Type               = \color [ HTML ] { 336666 } ,
1989 InitialValues           = \@@_piton:n ,
1990 Interpol.Inside         = { \l_@@_font_command_tl \@@_piton:n } ,
1991 TypeParameter           = \color [ HTML ] { 336666 } \itshape ,
1992 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).

```

1993 Identifier.Internal    = \@@_identifier:n ,
1994 Identifier              = ,
1995 Directive               = \color [ HTML ] { AA6600 } ,
1996 Tag                     = \colorbox { gray!10 } ,
1997 UserFunction            = \PitonStyle { Identifier } ,
1998 Prompt                 = ,
1999 Discard                 = \use_none:n
2000 }

```

2.10 Styles specific to the language expl

```

2001 \clist_new:N \g_@@_expl_styles_clist
2002 \clist_gset:Nn \g_@@_expl_styles_clist
2003 {
2004   Scope.l ,
2005   Scope.g ,
2006   Scope.c
2007 }
2008 \clist_map_inline:Nn \g_@@_expl_styles_clist
2009 {
2010   \keys_define:nn { piton / Styles }
2011   {
2012     #1 .value_required:n = true ,
2013     #1 .code:n =
2014       \tl_set:cn
2015       {
2016         pitonStyle _
2017         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
2018         { \l_@@_SetPitonStyle_option_str _ }
2019         #1
2020       }
2021     { ##1 }

```

```

2022     }
2023 }
2024 \SetPitonStyle [ expl ]
2025 {
2026     Scope.l           = ,
2027     Scope.g           = \bfseries ,
2028     Scope.c           = \slshape ,
2029     Type.bool         = \color [ HTML ] { AA6600 } ,
2030     Type.box          = \color [ HTML ] { 267910 } ,
2031     Type.clist        = \color [ HTML ] { 309030 } ,
2032     Type.fp           = \color [ HTML ] { FF3300 } ,
2033     Type.int          = \color [ HTML ] { FF6600 } ,
2034     Type.seq          = \color [ HTML ] { 309030 } ,
2035     Type.skip         = \color [ HTML ] { 0CC060 } ,
2036     Type.str          = \color [ HTML ] { CC3300 } ,
2037     Type.tl           = \color [ HTML ] { AA2200 } ,
2038     Module.bool       = \color [ HTML ] { AA6600 } ,
2039     Module.box        = \color [ HTML ] { 267910 } ,
2040     Module.cs         = \bfseries \color [ HTML ] { 006699 } ,
2041     Module.exp        = \bfseries \color [ HTML ] { 404040 } ,
2042     Module.hbox       = \color [ HTML ] { 267910 } ,
2043     Module.prg        = \bfseries ,
2044     Module.clist      = \color [ HTML ] { 309030 } ,
2045     Module.fp         = \color [ HTML ] { FF3300 } ,
2046     Module.int        = \color [ HTML ] { FF6600 } ,
2047     Module.seq        = \color [ HTML ] { 309030 } ,
2048     Module.skip       = \color [ HTML ] { 0CC060 } ,
2049     Module.str        = \color [ HTML ] { CC3300 } ,
2050     Module.tl         = \color [ HTML ] { AA2200 } ,
2051     Module.vbox       = \color [ HTML ] { 267910 }
2052 }

```

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)).

```

2053 \hook_gput_code:nnn { begindocument } { . }
2054 {
2055     \bool_if:NT \g_@@_math_comments_bool
2056     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
2057 }

```

2.11 Highlighting some identifiers

```

2058 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
2059 {
2060     \clist_set:Nn \l_tmpa_clist { #2 }
2061     \tl_if_novalue:nTF { #1 }
2062     {
2063         \clist_map_inline:Nn \l_tmpa_clist
2064         { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
2065     }
2066     {
2067         \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
2068         \str_if_eq:onT \l_tmpa_str { current-language }
2069         { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
2070         \clist_map_inline:Nn \l_tmpa_clist
2071         { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
2072     }
2073 }
2074 \cs_new_protected:Npn \@@_identifier:n #1
2075 {
2076     \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ #1 }

```

```

2077     {
2078       \cs_if_exist_use:cF { PitonIdentifier _ #1 }
2079       { \PitonStyle { Identifier } }
2080     }
2081     { #1 }
2082   }

```

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`).

```

2083 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
2084 {

```

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

```

2085   { \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}`.

```

2086   \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ #1 }
2087   { \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`.**

```

2088   \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
2089   { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
2090   \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.

```

2091   \seq_if_in:NoF { g_@@_languages_seq { \l_piton_language_str }
2092   { \seq_gput_left:No { g_@@_languages_seq { \l_piton_language_str } }
2093   }

```

```

2094 \NewDocumentCommand \PitonClearUserFunctions { ! o }
2095 {
2096   \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.

```

2097   { \@@_clear_all_functions: }
2098   { \@@_clear_list_functions:n { #1 } }
2099 }

```

```

2100 \cs_new_protected:Npn \@@_clear_list_functions:n #1
2101 {
2102   \clist_set:Nn \l_tmpa_clist { #1 }
2103   \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
2104   \clist_map_inline:nn { #1 }
2105   { \seq_gremove_all:Nn { g_@@_languages_seq { ##1 } }
2106   }

```

```

2107 \cs_new_protected:Npn \@@_clear_functions_i:n #1
2108 { \@@_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).

```

2109 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2110 {
2111   \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2112   {
2113     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2114     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }

```

```

2115     \seq_gclear:c { g_@@_functions _ #1 _ seq }
2116   }
2117 }
2118 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }

2119 \cs_new_protected:Npn \@@_clear_functions:n #1
2120 {
2121   \@@_clear_functions_i:n { #1 }
2122   \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2123 }

```

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

```

2124 \cs_new_protected:Npn \@@_clear_all_functions:
2125 {
2126   \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2127   \seq_gclear:N \g_@@_languages_seq
2128 }

```

```

2129 \AtEndDocument
2130 {

```

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

```

2131   \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.

```

2132   \IfPDFManagementActiveTF
2133   { \@@_join_files: }
2134   { \@@_join_files_legacy: }
2135 }

```

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

```

2136 \cs_new_protected:Npn \@@_join_files:
2137 {
2138   \seq_map_inline:Nn \g_@@_join_seq
2139   {
2140     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2141     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2142     \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2143     {
2144       <<
2145         /Type /Filespec
2146         /UF <\l_tmpa_str>
2147         /EF << /F~\pdf_object_ref_last: >>
2148         /Desc (Computer~listing)
2149         /AFRelationship /Supplement
2150       >>
2151     }
2152   }
2153 }

```

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.

```

2154 \cs_new_protected:Npn \@@_join_files_legacy:
2155 {
2156   \seq_map_inline:Nn \g_@@_join_seq
2157   {
2158     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2159     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2160     \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`.

```

2161     {
2162         /Subtype /FileAttachment
2163         /F~2
2164         /Name /Paperclip
2165         /Contents (Computer~listing)
2166         /FS <<
2167             /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).

```

2168         /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`).

```

2169         /EF << /F~\pdf_object_ref_last: >>
2170         /AFRelationship /Supplement
2171     >>
2172 }
2173 }
2174 }

```

2.12 Spaces of indentation

```

2175 \cs_new_protected:Npn \@@_define_leading_space_normal:
2176 {
2177     \cs_set_protected:Npn \@@_leading_space:
2178     {
2179         \int_gincr:N \g_@@_indentation_int

```

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

```

2180         \hbox:n { ~ }
2181     }
2182 }
2183 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2184 {
2185     \cs_set_protected:Npn \@@_leading_space:
2186     {
2187         \int_gincr:N \g_@@_indentation_int
2188         \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2189         {
2190             \color { white }
2191             \transparent { 0 }
2192             . % previously : □ U+2423
2193         }
2194         \pdfextension literal { EMC }
2195     }
2196 }
2197 \@@_define_leading_space_Foxit:

```

2.13 Security

```

2198 \AddToHook { env / piton / before }
2199 { \@@_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.

```

2200 \cs_new_protected:Npn \@@_unknown_key:nn #1 #2
2201 {
2202   \str_set_eq:NN \l_tmpa_str \l_keys_key_str
2203   \str_replace_all:Nnn \l_tmpa_str { ~ } { - }
2204   \str_set:Nx \l_tmpa_str { \str_lowercase:f { \l_tmpa_str } }
2205   \bool_set_false:N \l_tmpa_bool
2206   \clist_map_inline:nn { #1 }
2207   {
2208     \keys_if_exist:neT { ##1 } { \l_tmpa_str }
2209     {
2210       \@@_error:n { key~with~normal~form~exists }
2211       \bool_set_true:N \l_tmpa_bool
2212       \clist_map_break:
2213     }
2214   }
2215   \bool_if:NF \l_tmpa_bool { \@@_error:n { #2 } }
2216 }
2217 \@@_msg_new:nn { key~with~normal~form~exists }
2218 {
2219   The~key~'\l_keys_key_str'~does~not~exists.~It~will~be~ignored.\\
2220   Maybe~you~want~to~use~the~key~'\l_tmpa_str'.
2221 }
2222 \@@_msg_new:nn { No~environment~piton }
2223 {
2224   There~is~no~environment~piton!\\
2225   There~is~an~environment~{Piton}~and~a~command~
2226   \token_to_str:N \piton\ but~there~is~no~environment~
2227   {piton}.~This~error~is~fatal.
2228 }
2229 \@@_msg_new:nn { rounded~corners~without~Tikz }
2230 {
2231   TikZ~not~used \\
2232   You~can't~use~the~key~'rounded~corners'~because~
2233   you~have~not~loaded~the~package~TikZ. \\
2234   If~you~go~on,~that~key~will~be~ignored. \\
2235   You~won't~have~similar~error~till~the~end~of~the~document.
2236 }
2237 \@@_msg_new:nn { tcolorbox~not~loaded }
2238 {
2239   tcolorbox~not~loaded \\
2240   You~can't~use~the~key~'tcolorbox'~because~
2241   you~have~not~loaded~the~package~tcolorbox. \\
2242   Use~\token_to_str:N \usepackage[breakable]{tcolorbox}. \\
2243   If~you~go~on,~that~key~will~be~ignored.
2244 }
2245 \@@_msg_new:nn { library~breakable~not~loaded }
2246 {
2247   breakable~not~loaded \\
2248   You~can't~use~the~key~'tcolorbox'~because~
2249   you~have~not~loaded~the~library~'breakable'~of~tcolorbox'. \\
2250   Use~\token_to_str:N \tcbuselibrary{breakable}~in~the~preamble~
2251   of~your~document.\\
2252   If~you~go~on,~that~key~will~be~ignored.
2253 }
2254 \@@_msg_new:nn { Language~not~defined }
2255 {
2256   Language~not~defined \\
2257   The~language~'\l_tmpa_tl'~has~not~been~defined~previously.\\

```

```

2258     If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2259     will~be~ignored.
2260 }

2261 \@@_msg_new:nn { bad~version~of~piton.lua }
2262 {
2263     Bad~number~version~of~'piton.lua'\
2264     The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2265     version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2266     address~that~issue.
2267 }

2268 \@@_msg_new:nn { Unknown~key~NewPitonLanguage }
2269 {
2270     Unknown~key~for~\token_to_str:N \NewPitonLanguage.\
2271     The~key~'\l_keys_key_str'~is~unknown.\
2272     This~key~will~be~ignored.\
2273 }

2274 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
2275 {
2276     The~style~'\l_keys_key_str'~is~unknown.\
2277     This~setting~will~be~ignored.\
2278     The~available~styles~are~(in~alphabetic~order):~
2279     \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2280 }

2281 \@@_msg_new:nn { Invalid~key }
2282 {
2283     Wrong~use~of~key.\
2284     You~can't~use~the~key~'\l_keys_key_str'~here.\
2285     That~key~will~be~ignored.
2286 }

2287 \@@_msg_new:nn { Unknown~key~for~line~numbers }
2288 {
2289     Unknown~key. \
2290     The~key~'line~numbers / \l_keys_key_str'~is~unknown.\
2291     The~available~keys~of~the~family~'line~numbers'~are~(in~
2292     alphabetic~order):~
2293     absolute,~false,~label~empty~lines,~position,~resume,~skip~empty~lines,~
2294     sep,~start~and~true.\
2295     That~key~will~be~ignored.
2296 }

2297 \@@_msg_new:nn { Unknown~key~for~marker }
2298 {
2299     Unknown~key. \
2300     The~key~'marker / \l_keys_key_str'~is~unknown.\
2301     The~available~keys~of~the~family~'marker'~are~(in~
2302     alphabetic~order):~ beginning,~end~and~include~lines.\
2303     That~key~will~be~ignored.
2304 }

2305 \@@_msg_new:nn { bad~range~specification }
2306 {
2307     Incompatible~keys.\
2308     You~can't~specify~the~range~of~lines~to~include~by~using~both~
2309     markers~and~explicit~number~of~lines.\
2310     Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2311 }

2312 \cs_new_nopar:Nn \@@_thepage:
2313 {
2314     \thepage
2315     \cs_if_exist:NT \insertframenumber
2316     {
2317         ~(frame~\insertframenumber

```

```

2318     \cs_if_exist:NT \beamer@slidenumber { ,~slide~\insertslidenumber }
2319   )
2320 }
2321 }

```

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`.

```

2322 \@@_msg_new:nn { SyntaxError }
2323 {
2324   Syntax~Error~on~page~\@@_thepage:.\
2325   Your~code~of~the~language~'\l_piton_language_str'~is~not~
2326   syntactically~correct.\
2327   It~won't~be~printed~in~the~PDF~file.
2328 }
2329 \@@_msg_new:nn { FileError }
2330 {
2331   File~Error.\
2332   It's~not~possible~to~write~on~the~file~'#1' \
2333   \sys_if_shell_unrestricted:F
2334   { (try~to~compile~with~'lualatex--shell-escape').\ }
2335   If~you~go~on,~nothing~will~be~written~on~that~file.
2336 }
2337 \@@_msg_new:nn { InexistentDirectory }
2338 {
2339   Inexistent~directory.\
2340   The~directory~'\l_@@_path_write_str'~
2341   given~in~the~key~'path-write'~does~not~exist.\
2342   Nothing~will~be~written~on~'\l_@@_write_str'.
2343 }
2344 \@@_msg_new:nn { begin~marker~not~found }
2345 {
2346   Marker~not~found.\
2347   The~range~'\l_@@_begin_range_str'~provided~to~the~
2348   command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
2349   The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
2350 }
2351 \@@_msg_new:nn { end~marker~not~found }
2352 {
2353   Marker~not~found.\
2354   The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
2355   provided~to~the~command~\token_to_str:N \PitonInputFile\
2356   has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
2357   be~inserted~till~the~end.
2358 }
2359 \@@_msg_new:nn { Unknown~file }
2360 {
2361   Unknown~file. \
2362   The~file~'#1'~is~unknown.\
2363   Your~command~\token_to_str:N \PitonInputFile\ will~be~discarded.
2364 }
2365 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2366 {
2367   \bool_if:NF \g_@@_beamer_bool
2368   { \@@_error_or_warning:n { Without~beamer } }
2369 }
2370 \@@_msg_new:nn { Without~beamer }
2371 {
2372   Key~'\l_keys_key_str'~without~Beamer.\
2373   You~should~not~use~the~key~'\l_keys_key_str'~since~you~
2374   are~not~in~Beamer.\

```

```

2375     However,~you~can~go~on.
2376 }
2377 \@@_msg_new:nn { rowcolor~in~detected-commands }
2378 {
2379     'rowcolor'~forbidden~in~'detected-commands'.\\
2380     You~should~put~'rowcolor'~in~'raw-detected-commands'.\\
2381     That~key~will~be~ignored.
2382 }
2383 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2384 {
2385     Unknown~key. \\
2386     The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
2387     It~will~be~ignored.\\
2388     For~a~list~of~the~available~keys,~type~H~<return>.
2389 }
2390 {
2391     The~available~keys~are~(in~alphabetic~order):~
2392     annotation,~
2393     add-to-split-separation,~
2394     auto-gobble,~
2395     background-color,~
2396     begin-range,~
2397     box,~
2398     break-lines,~
2399     break-lines-in-piton,~
2400     break-lines-in-Piton,~
2401     break-numbers-anywhere,~
2402     break-strings-anywhere,~
2403     continuation-symbol,~
2404     continuation-symbol-on-indentation,~
2405     detected-beamer-commands,~
2406     detected-beamer-environments,~
2407     detected-commands,~
2408     end-of-broken-line,~
2409     end-range,~
2410     env-gobble,~
2411     env-used-by-split,~
2412     font-command(+),~
2413     gobble,~
2414     indent-broken-lines,~
2415     join,~
2416     label-as-zlabel,~
2417     language,~
2418     left-margin,~
2419     line-numbers/,~
2420     marker/,~
2421     math-comments,~
2422     no-join,~
2423     no-write,~
2424     path,~
2425     path-write,~
2426     print,~
2427     prompt-background-color,~
2428     raw-detected-commands,~
2429     resume,~
2430     right-margin,~
2431     rounded-corners,~
2432     show-spaces,~
2433     show-spaces-in-strings,~
2434     splittable,~
2435     splittable-on-empty-lines,~
2436     split-on-empty-lines,~
2437     split-separation,~

```

```

2438     tabs-auto-gobble,~
2439     tab-size,~
2440     tcolorbox,~
2441     varwidth,~
2442     vertical-detected-commands,~
2443     width-and-write.
2444 }

2445 \@@_msg_new:nn { label-with-lines-numbers }
2446 {
2447     You~can't~use~the~command~\token_to_str:N \label\
2448     or~\token_to_str:N \zlabel\ because~the~key~'line-numbers'
2449     ~is~not~active.\\
2450     If~you~go~on,~that~command~will~ignored.
2451 }

2452 \@@_msg_new:nn { overlay-without-beamer }
2453 {
2454     You~can't~use~an~argument~<...>~for~your~command~
2455     \token_to_str:N \PitonInputFile\ because~you~are~not~
2456     in~Beamer.\\
2457     If~you~go~on,~that~argument~will~be~ignored.
2458 }

2459 \@@_msg_new:nn { label-as-zlabel-needs-zref-package }
2460 {
2461     The~key~'label-as-zlabel'~requires~the~package~'zref'.~
2462     Please~load~the~package~'zref'~before~setting~the~key.\\
2463     This~error~is~fatal.
2464 }
2465 \hook_gput_code:nnn { begindocument } { . }
2466 {
2467     \bool_if:NT \g_@@_label_as_zlabel_bool
2468     {
2469         \IfPackageLoadedF { zref-base }
2470         { \@@_fatal:n { label-as-zlabel-needs-zref-package } }
2471     }
2472 }

```

2.15 We load piton.lua

```

2473 \cs_new_protected:Npn \@@_test_version:n #1
2474 {
2475     \str_if_eq:onF \PitonFileVersion { #1 }
2476     { \@@_error:n { bad~version~of~piton.lua } }
2477 }

2478 \hook_gput_code:nnn { begindocument } { . }
2479 {
2480     \lua_load_module:n { piton }
2481     \lua_now:n
2482     {
2483         tex.sprint ( luatexbase.catcodetables.expl ,
2484                     [[\@@_test_version:n {}] .. piton_version .. "]" )
2485     }
2486 }

```

</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`.

```
2487  $\langle$ *LUA $\rangle$ 
2488 piton.comment_latex = piton.comment_latex or ">"
2489 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`.

```
2490 piton.write_files = { }
2491 piton.join_files = { }

2492 local sprintL3
2493 function sprintL3 ( s )
2494   tex.sprint ( luatexbase.catcodetables.expl , s )
2495 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.

```
2496 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2497 local Cg, Cmt, Cb = lpeg.Cg, lpeg.Cmt, lpeg.Cb
2498 local B, R = lpeg.B, lpeg.R
```

The following line is mandatory.

```
2499 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”).

```
2500 local Q
2501 function Q ( pattern )
2502   return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2503 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.

```
2504 local L
2505 function L ( pattern ) return
2506   Ct ( C ( pattern ) )
2507 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.

```

2508 local Lc
2509 function Lc ( string ) return
2510   Cc ( { luatexbase.catcodetables.expl , string } )
2511 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)

```

2512 local K
2513 function K ( style , pattern ) return
2514   Lc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] )
2515   * Q ( pattern )
2516   * Lc "}"
2517 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.

```

2518 local WithStyle
2519 function WithStyle ( style , pattern ) return
2520   Ct ( Cc "Open" * Cc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] * Cc "}" )
2521   * pattern
2522   * Ct ( Cc "Close" )
2523 end

```

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

```

2524 Escape = P ( false )
2525 EscapeClean = P ( false )
2526 if piton.begin_escape then
2527   Escape =
2528     P ( piton.begin_escape )
2529     * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
2530     * 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).

```

2531   EscapeClean =
2532     P ( piton.begin_escape )
2533     * ( 1 - P ( piton.end_escape ) ) ^ 1
2534     * P ( piton.end_escape )
2535 end

2536 EscapeMath = P ( false )
2537 if piton.begin_escape_math then
2538   EscapeMath =
2539     P ( piton.begin_escape_math )
2540     * Lc "$"
2541     * L ( ( 1 - P ( piton.end_escape_math ) ) ^ 1 )
2542     * Lc "$"
2543     * P ( piton.end_escape_math )
2544 end

```


The basic syntactic LPEG

```
2545 local alpha , digit = lpeg.alpha , lpeg.digit
2546 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.

```
2547 local letter = alpha + "_" + "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "í"
2548               + "ô" + "û" + "ü" + "Å" + "Ä" + "Ç" + "É" + "È" + "Ê" + "Ë"
2549               + "Ī" + "Ĭ" + "Ō" + "Ū" + "Ū"
2550
2551 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).

```
2552 local identifier = letter * alphanum ^ 0
```

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

```
2553 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.³

```
2554 local allow_underscores_except_first
2555 function allow_underscores_except_first ( p )
2556     return p * (P "_" + p)^0
2557 end
2558 local allow_underscores
2559 function allow_underscores ( p )
2560     return (P "_" + p)^0
2561 end
2562 local digits_to_number
2563 function digits_to_number(prefix, digits)
2564     -- The edge cases of what is allowed in number literals is modelled after
2565     -- OCaml numbers, which seems to be the most permissive language
2566     -- in this regard (among C, OCaml, Python & SQL).
2567     return prefix
2568         * allow_underscores_except_first(digits^1)
2569         * (P "." * #(1 - P ".") * allow_underscores(digits))^~1
2570         * (S "eE" * S "+-")^~1 * allow_underscores_except_first(digits^1))^~1
2571 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).

```
2572 local Number =
2573     K ( 'Number.Internal' ,
2574         digits_to_number (P "0x" + P "0X", R "af" + R "AF" + digit)
2575         + digits_to_number (P "0o" + P "0O", R "07")
2576         + digits_to_number (P "0b" + P "0B", R "01")
2577         + digits_to_number ( "" , digit )
2578     )
```

³The 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?).

```
2579 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`.

```
2580 if piton.begin_escape then
2581   lpeg_central = lpeg_central - piton.begin_escape
2582 end
2583 if piton.begin_escape_math then
2584   lpeg_central = lpeg_central - piton.begin_escape_math
2585 end
2586 local Word = Q ( lpeg_central ^ 1 )
```

```
2587 local Space = Q " " ^ 1
2588 local SkipSpace = Q " " ^ 0
2589
2590 local Punct = Q ( S ".,:;!" )
2591
2592 local Tab = "\t" * Lc [[ \@_tab: ]]
```

```
2593 local LeadingSpace = Lc [[ \@_leading_space: ]] * P " "
```

```
2594 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.

```
2595 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).

```
2596 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
2597 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
2598 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
2599 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.

```
2600 local detectedCommands = P ( false )
2601 for _ , x in ipairs ( detected_commands ) do
2602   detectedCommands = detectedCommands + P ( "\\\" .. x )
2603 end
```

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

```

2604 local rawDetectedCommands = P ( false )
2605 for _ , x in ipairs ( raw_detected_commands ) do
2606   rawDetectedCommands = rawDetectedCommands + P ( "\\\" .. x )
2607 end
2608 local beamerCommands = P ( false )
2609 for _ , x in ipairs ( beamer_commands ) do
2610   beamerCommands = beamerCommands + P ( "\\\" .. x )
2611 end
2612 local beamerEnvironments = P ( false )
2613 for _ , x in ipairs ( beamer_environments ) do
2614   beamerEnvironments = beamerEnvironments + P ( x )
2615 end

```

Several tools for the construction of the main LPEG

```

2616 local LPEG0 = { }
2617 local LPEG1 = { }
2618 local LPEG2 = { }
2619 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*.

```

2620 local Compute_braces
2621 function Compute_braces ( lpeg_string ) return
2622   P { "E" ,
2623     E =
2624     (
2625       "{" * V "E" * "}"
2626       +
2627       lpeg_string
2628       +
2629       ( 1 - S "{" )
2630     ) ^ 0
2631   }
2632 end

```

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

```

2633 local Compute_DetectedCommands
2634 function Compute_DetectedCommands ( lang , braces ) return
2635   Ct (
2636     Cc "Open"
2637     * C ( detectedCommands * space ^ 0 * P "{" )
2638     * Cc "}"
2639   )
2640   * ( braces
2641     / ( function ( s )
2642         if s ~= '' then return
2643           LPEG1[lang] : match ( s )
2644         end
2645       end )
2646   )
2647   * P "}"
2648   * Ct ( Cc "Close" )
2649 end

```

```

2650 local Compute_RawDetectedCommands
2651 function Compute_RawDetectedCommands ( lang , braces ) return
2652   Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
2653 end

2654 local Compute_LPEG_cleaner
2655 function Compute_LPEG_cleaner ( lang , braces ) return
2656   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
2657         * ( braces
2658           / ( function ( s )
2659               if s ~= '' then return
2660                 LPEG_cleaner[lang] : match ( s )
2661               end
2662             end )
2663         )
2664         * "}"
2665         + EscapeClean
2666         + C ( P ( 1 ) )
2667         ) ^ 0 ) / table.concat
2668 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`.

```

2669 local ParseAgain
2670 function ParseAgain ( code )
2671   if code ~= '' then return

```

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

```

2672   LPEG1[piton.language] : match ( code )
2673 end
2674 end

```

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

```

2675 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.

```

2676 local Compute_Beamer
2677 function Compute_Beamer ( lang , braces )

```

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

```

2678   local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
2679   lpeg = lpeg +
2680     Ct ( Cc "Open"
2681         * C ( beamerCommands
2682             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2683             * P "{"
2684           )
2685         * Cc "}"
2686       )
2687     * ( braces /
2688       ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2689     * "}"
2690     * Ct ( Cc "Close" )

```

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

```

2691 lpeg = lpeg +
2692   L ( P [[\alt]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2693   * ( braces /
2694     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2695   * L ( P "}" )
2696   * ( braces /
2697     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2698   * L ( P "}" )

```

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

```

2699 lpeg = lpeg +
2700   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2701   * ( braces
2702     / ( function ( s )
2703       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2704   * L ( P "}" )
2705   * ( braces
2706     / ( function ( s )
2707       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2708   * L ( P "}" )
2709   * ( braces
2710     / ( function ( s )
2711       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2712   * L ( P "}" )

```

Now, the environments of Beamer.

```

2713 for _ , x in ipairs ( beamer_environments ) do
2714   lpeg = lpeg +
2715     Ct ( Cc "Open"
2716       * C (
2717         P ( [[\begin{]] .. x .. "]" )
2718         * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2719       )
2720       * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
2721       * Cc ( [[\end{]] .. x .. "]" )
2722     )
2723   * (

```

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).

```

2724     (
2725       P { "E" ,
2726         E = (
2727           P ( [[\begin{]] .. x .. "]" )
2728           * V "E"
2729           * P ( [[\end{]] .. x .. "]" )
2730         +
2731         (
2732           1
2733           - P ( [[\begin{]] .. x .. "]" )
2734           - P ( [[\end{]] .. x .. "]" )
2735         )
2736       ) ^ 0
2737     }
2738   )
2739   / ( function ( s )
2740     if s ~= '' then return
2741       LPEG1[lang] : match ( s )
2742     end
2743   end )
2744 )

```

```

2745         * P ( [[\end{}} .. x .. "}" )
2746         * Ct ( Cc "Close" )
2747     end

```

Now, you can return the value we have computed.

```

2748     return lpeg
2749 end

```

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

```

2750 local CommentMath =
2751     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`.

```

2752 local Prompt =
2753     K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
2754     * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

2755 local EOL =
2756     P "\r"
2757     *
2758     (
2759         space ^ 0 * -1
2760         +
2761         Cc "EOL"
2762     )
2763     * ( LeadingSpace ^ 0 * # ( 1 - S " \r" ) ) ^ -1

```

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

```

2764 local CommentLaTeX =
2765     P ( piton.comment_latex )
2766     * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}}]
2767     * L ( ( 1 - P "\r" ) ^ 0 )
2768     * Lc "}}"
2769     * ( 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).

```

2770 --python Python
2771 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.

```

2772 local Operator =
2773     K ( 'Operator' ,
2774         P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "://" + "*"
2775         + S "--+/*%=<>&.@|" )
2776
2777 local OperatorWord =
2778     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`.

```

2779 local For = K ( 'Keyword' , P "for" )
2780         * Space
2781         * Identifier
2782         * Space
2783         * K ( 'Keyword' , P "in" )
2784
2785 local Keyword =
2786   K ( 'Keyword' ,
2787     P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
2788     "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
2789     "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
2790     "try" + "while" + "with" + "yield" + "yield from" )
2791   + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
2792
2793 local Builtin =
2794   K ( 'Name.Builtin' ,
2795     P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
2796     "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
2797     "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
2798     "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
2799     "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
2800     "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
2801     + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
2802     "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
2803     "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
2804     "vars" + "zip" )
2805
2806 local Exception =
2807   K ( 'Exception' ,
2808     P "ArithmeticError" + "AssertionError" + "AttributeError" +
2809     "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
2810     "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
2811     "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
2812     "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
2813     "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
2814     "NotImplementedError" + "OSError" + "OverflowError" +
2815     "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
2816     "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
2817     "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
2818     + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
2819     "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
2820     "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
2821     "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
2822     "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
2823     "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
2824     "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
2825     "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
2826     "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
2827     "RecursionError" )
2828
2829 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.

```

2830 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:`

```

2831 local DefClass =
2832   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`

```

2833 local ImportAs =
2834   K ( 'Keyword' , "import" )
2835   * Space
2836   * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
2837   * (
2838     ( Space * K ( 'Keyword' , "as" ) * Space
2839       * K ( 'Name.Namespace' , identifier ) )
2840     +
2841     ( SkipSpace * Q "," * SkipSpace
2842       * K ( 'Name.Namespace' , identifier ) ) ^ 0
2843   )

```

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`

```

2844 local FromImport =
2845   K ( 'Keyword' , "from" )
2846   * Space * K ( 'Name.Namespace' , identifier )
2847   * 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 instruction⁴ in that interpolation:

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

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

```

2848 local PercentInterpol =
2849   K ( 'String.Interpol' ,
2850     P "%"

```

⁴There 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`.


```

2851 * ( "(" * alphanum ^ 1 * ")" ) ^ -1
2852 * ( S "-#0 +" ) ^ 0
2853 * ( digit ^ 1 + "*" ) ^ -1
2854 * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
2855 * ( S "HLL" ) ^ -1
2856 * S "sdfFeExXorgiGauc%"
2857 )

```

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.⁵

```

2858 local SingleShortString =
2859   WithStyle ( 'String.Short.Internal' ,

```

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

```

2860   Q ( P "f'" + "F'" )
2861   * (
2862     K ( 'String.Interpol' , "{" )
2863     * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0 )
2864     * Q ( P ":" * ( 1 - S "}':" ) ^ 0 ) ^ -1
2865     * K ( 'String.Interpol' , "}" )
2866     +
2867     SpaceInString
2868     +
2869     Q ( ( P "\\'" + "\\\\" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
2870   ) ^ 0
2871   * Q ""
2872 +

```

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

```

2873   Q ( P "'" + "r'" + "R'" )
2874   * ( Q ( ( P "\\'" + "\\\\" + 1 - S " 'r%" ) ^ 1 )
2875     + SpaceInString
2876     + PercentInterpol
2877     + Q "%"
2878   ) ^ 0
2879   * Q "" )

2880 local DoubleShortString =
2881   WithStyle ( 'String.Short.Internal' ,
2882     Q ( P "f\"" + "F\"" )
2883     * (
2884       K ( 'String.Interpol' , "{" )
2885       * K ( 'Interpol.Inside' , ( 1 - S "}\" )" ) ^ 0 )
2886       * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}\" )" ) ^ 0 ) ) ^ -1
2887       * K ( 'String.Interpol' , "}" )
2888       +
2889       SpaceInString
2890       +
2891       Q ( ( P "\\\"" + "\\\\" + "{{" + "}}" + 1 - S " {}\" )" ^ 1 )
2892     ) ^ 0
2893     * Q "\"
2894   +
2895     Q ( P "\" + "r\"" + "R\"" )
2896     * ( Q ( ( P "\\\"" + "\\\\" + 1 - S " \"r%" ) ^ 1 )
2897       + SpaceInString
2898       + PercentInterpol
2899       + Q "%"
2900     ) ^ 0
2901     * Q "\" )

```

⁵The 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`.

```

2902
2903     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.

```

2904     local braces =
2905         Compute_braces
2906         (
2907             ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
2908             * ( P '\\\"' + 1 - S "\"" ) ^ 0 * "\""
2909         +
2910             ( P '\'' + 'r\'' + 'R\'' + 'f\'' + 'F\'' )
2911             * ( P '\\\'' + 1 - S '\'' ) ^ 0 * '\''
2912         )
2913
2914     if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

Detected commands

```

2915     DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
2916     + Compute_RawDetectedCommands ( 'python' , braces )

```

LPEG_cleaner

```

2917     LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

The long strings

```

2918     local SingleLongString =
2919         WithStyle ( 'String.Long.Internal' ,
2920             ( Q ( S "fF" * P "'''" )
2921                 * (
2922                     K ( 'String.Interpol' , "{" )
2923                     * K ( 'Interpol.Inside' , ( 1 - S "};\r" - "'''" ) ^ 0 )
2924                     * Q ( P ":" * ( 1 - S "};\r" - "'''" ) ^ 0 ) ^ -1
2925                     * K ( 'String.Interpol' , "}" )
2926                 +
2927                     Q ( ( 1 - P "'''" - S "{'}\r" ) ^ 1 )
2928                 +
2929                     EOL
2930             ) ^ 0
2931         +
2932             Q ( ( S "rR" ) ^ -1 * "'''" )
2933             * (
2934                 Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
2935                 +
2936                 PercentInterpol
2937                 +
2938                 P "%"
2939                 +
2940                 EOL
2941             ) ^ 0
2942         )
2943         * Q "'''" )

```

```

2944 local DoubleLongString =
2945   WithStyle ( 'String.Long.Internal' ,
2946     (
2947       Q ( S "fF" * "\"\\\"" )
2948       * (
2949         K ( 'String.Interpol', "{ " )
2950         * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "\"\\\"" ) ^ 0 )
2951         * Q ( ":" * (1 - S "}:\\r" - "\"\\\"" ) ^ 0 ) ^ -1
2952         * K ( 'String.Interpol' , "}" )
2953         +
2954         Q ( ( 1 - S "{}\\r" - "\"\\\"" ) ^ 1 )
2955         +
2956         EOL
2957       ) ^ 0
2958       +
2959       Q ( S "rR" ^ -1 * "\"\\\"" )
2960       * (
2961         Q ( ( 1 - P "\"\\\"" - S "%\\r" ) ^ 1 )
2962         +
2963         PercentInterpol
2964         +
2965         P "%"
2966         +
2967         EOL
2968       ) ^ 0
2969     )
2970     * Q "\"\\\""
2971   )
2972 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`).

```

2973 local StringDoc =
2974   K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\\\"" )
2975   * ( K ( 'String.Doc.Internal' , (1 - P "\"\\\"" - "\\r" ) ^ 0 ) * EOL
2976     * Tab ^ 0
2977   ) ^ 0
2978   * 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.

```

2979 local Comment =
2980   WithStyle
2981     ( 'Comment.Internal' ,
2982       Q "#" * ( CommentMath + Q ( ( 1 - S "$\\r" ) ^ 1 ) ) ^ 0 -- $
2983     )
2984   * ( 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).

```

2985 local expression =
2986   P { "E" ,
2987     E = (
2988       "'"' * ( P "\\'" + 1 - S "'\\r" ) ^ 0 * "'"
2989       + "\"\" * ( P "\\\"" + 1 - S "\"\\r" ) ^ 0 * "\""
2990       + "{" * V "F" * "}"
2991       + "(" * V "F" * ")"

```

```

2991         + "[" * V "F" * "]"
2992         + ( 1 - S "{ } ( [ ] \r , " ) ) ^ 0 ,
2993     F = (   "{" * V "F" * "}"
2994           + "(" * V "F" * ")"
2995           + "[" * V "F" * "]"
2996           + ( 1 - S "{ } ( [ ] \r \'"' " ) ) ^ 0
2997     }

```

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`.

```

2998     local Params =
2999     P { "E" ,
3000         E = ( V "F" * ( Q " , " * V "F" ) ^ 0 ) ^ -1 ,
3001         F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
3002           * ( Q ":" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3003           * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
3004     }

```

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`...

```

3005     local DefFunction =
3006     K ( 'Keyword' , "def" )
3007     * Space
3008     * K ( 'Name.Function.Internal' , identifier )
3009     * SkipSpace
3010     * Q "(" * Params * Q ")"
3011     * SkipSpace
3012     * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3013     * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
3014     * Q ":"
3015     * ( SkipSpace
3016       * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
3017       * Tab ^ 0
3018       * SkipSpace
3019       * StringDoc ^ 0 -- there may be additional docstrings
3020     ) ^ -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

```

3021     local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

The main LPEG for the language Python

```
3022 local EndKeyword
3023     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3024     EscapeMath + -1
```

First, the main loop :

```
3025 local Main =
3026     space ^ 0 * EOL -- faut-il le mettre en commentaire ?
3027     + Space
3028     + Tab
3029     + Escape + EscapeMath
3030     + Beamer
3031     + CommentLaTeX
3032     + DetectedCommands
3033     + Prompt
3034     + LongString
3035     + Comment
3036     + ExceptionInConsole
3037     + Delim
3038     + Operator
3039     + OperatorWord * EndKeyword
3040     + ShortString
3041     + Punct
3042     + FromImport
3043     + RaiseException
3044     + DefFunction
3045     + DefClass
3046     + For
3047     + Keyword * EndKeyword
3048     + Decorator
3049     + Builtin * EndKeyword
3050     + Identifier
3051     + Number
3052     + Word
```

Here, we must not put local, of course.

```
3053 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:`⁶.

```
3054 LPEG2.python =
3055     Ct (
3056         ( space ^ 0 * "\r" ) ^ -1
3057         * Lc [[ \@@_begin_line: ]]
3058         * LeadingSpace ^ 0
3059         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3060         * -1
3061         * Lc [[ \@@_end_line: ]]
3062     )
```

End of the Lua scope for the language Python.

```
3063 end
```

⁶Remember 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).

```

3064 --ocaml Ocaml OCaml
3065 do

3066     local SkipSpace = ( Q " " + EOL ) ^ 0
3067     local Space = ( Q " " + EOL ) ^ 1

3068     local braces = Compute_braces ( '\'' * ( 1 - S "\"" ) ^ 0 * '\'' )

3069     if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
3070     DetectedCommands =
3071         Compute_DetectedCommands ( 'ocaml' , braces )
3072         + Compute_RawDetectedCommands ( 'ocaml' , braces )
3073     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.

```

3074     function Q ( pattern, strict )
3075         if strict ~= nil then
3076             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3077         else
3078             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3079                 + Beamer + DetectedCommands + EscapeMath + Escape
3080         end
3081     end

3082     local K
3083     function K ( style , pattern, strict ) return
3084         Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
3085         * Q ( pattern, strict )
3086         * Lc "}"
3087     end

3088     local WithStyle
3089     function WithStyle ( style , pattern ) return
3090         Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{ ]] .. style .. "}{" ) * Cc "}" )
3091         * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
3092         * Ct ( Cc "Close" )
3093     end

```

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

```

3094     local balanced_parens =
3095         P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }

```

The strings of OCaml

```

3096     local ocaml_string =
3097         P "\""
3098         * (
3099             P " "
3100             +
3101             P ( ( P '\\\'' + 1 - S " \r" ) ^ 1 )
3102             +
3103             EOL -- ?
3104         ) ^ 0
3105     * P "\""

```

```

3106 local String =
3107   WithStyle
3108     ( 'String.Long.Internal' ,
3109       Q "\""
3110       * (
3111         SpaceInString
3112         +
3113         Q ( ( P '\\\\' + 1 - S "\r" ) ^ 1 )
3114         +
3115         EOL
3116       ) ^ 0
3117       * Q "\""
3118     )

```

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.

```

3119 local ext = ( R "az" + "_" ) ^ 0
3120 local open = "{" * Cg ( ext , 'init' ) * "/"
3121 local close = "/" * C ( ext ) * "}"
3122 local closeeq =
3123   Cmt ( close * Cb ( 'init' ) ,
3124     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3125 local QuotedStringBis =
3126   WithStyle ( 'String.Long.Internal' ,
3127     (
3128       Space
3129       +
3130       Q ( ( 1 - S "\r" ) ^ 1 )
3131       +
3132       EOL
3133     ) ^ 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.

```

3134 local QuotedString =
3135   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3136   ( 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).

```

3137 local comment =
3138   P {
3139     "A" ,
3140     A = Q "(*"
3141       * ( V "A"
3142         + Q ( ( 1 - S "\r$\\" - "(" - ")" ) ^ 1 ) -- $
3143         + Q ( ocaml_string )
3144         + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3145         + EOL
3146       ) ^ 0
3147       * Q "*)"
3148   }
3149 local Comment = WithStyle ( 'Comment.Internal' , comment )

```

Some standard LPEG

```
3150 local Delim = Q ( P "[" + "]" + S "[]" )
3151 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.

```
3152 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.

```
3153 local Constructor =
3154   P "::"
```

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

```
3155 * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3156 +
3157 P "[]"
3158 * Lc ([{\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]])
3159 K ( 'Name.Constructor' ,
3160     Q "`" ^ -1 * cap_identifier
3161     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )
```

```
3162 local ModuleType = K ( 'Name.Type' , cap_identifier )
```

```
3163 local OperatorWord =
3164   K ( 'Operator.Word' ,
3165       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )
```

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

```
3166 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3167   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3168   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3169   "struct" + "type" + "val"
```

```
3170 local Keyword =
3171   K ( 'Keyword' ,
3172       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3173       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3174       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3175       + "virtual" + "when" + "while" + "with" )
3176   + K ( 'Keyword.Constant' , P "true" + "false" )
3177   + K ( 'Keyword.Governing' , governing_keyword )
```

```
3178 local EndKeyword
3179   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3180   + EscapeMath + -1
```

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

```
3181 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3182   - ( 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`.

```
3183 local Identifier = K ( 'Identifier.Internal' , identifier )
```


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

```

3184 local ocaml_char =
3185   P "" *
3186   (
3187     ( 1 - S "\\\" )
3188     + "\\\"
3189     * ( S "\\ntbr \"
3190         + digit * digit * digit
3191         + P "x" * ( digit + R "af" + R "AF" )
3192         * ( digit + R "af" + R "AF" )
3193         * ( digit + R "af" + R "AF" )
3194         + P "o" * R "03" * R "07" * R "07" )
3195   )
3196   * ""
3197 local Char =
3198   K ( 'String.Short.Internal', ocaml_char )

```

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

```

3199 local TypeParameter =
3200   K ( 'TypeParameter' ,
3201       "" * 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.

```

3202 local DotNotation =
3203   (
3204     K ( 'Name.Module' , cap_identifier )
3205     * Q "."
3206     * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3207     +
3208     Identifier
3209     * Q "."
3210     * K ( 'Name.Field' , identifier )
3211   )
3212   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

```

The records

```

3213 local expression_for_fields_type =
3214   P { "E" ,
3215       E = ( "{" * V "F" * "}"
3216           + "(" * V "F" * ")"
3217           + TypeParameter
3218           + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
3219       F = ( "{" * V "F" * "}"
3220           + "(" * V "F" * ")"
3221           + ( 1 - S "{}()[]\r\"" ) + TypeParameter ) ^ 0
3222   }

```

```

3223 local expression_for_fields_value =
3224   P { "E" ,
3225       E = ( "{" * V "F" * "}"
3226           + "(" * V "F" * ")"
3227           + "[" * V "F" * "]"
3228           + ocaml_string + ocaml_char
3229           + ( 1 - S "{}()[];" ) ) ^ 0 ,
3230       F = ( "{" * V "F" * "}"
3231           + "(" * V "F" * ")"
3232           + "[" * V "F" * "]"
3233           + ocaml_string + ocaml_char
3234           + ( 1 - S "{}()[]\\"" ) ) ^ 0
3235   }

```

```

3236 local OneFieldDefinition =
3237   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3238   * K ( 'Name.Field' , identifier ) * SkipSpace
3239   * Q ":" * SkipSpace
3240   * K ( 'TypeExpression' , expression_for_fields_type )
3241   * SkipSpace

```

```

3242 local OneField =
3243   K ( 'Name.Field' , identifier ) * SkipSpace
3244   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3245   * ( C ( expression_for_fields_value ) / ParseAgain )
3246   * SkipSpace

```

The records.

```

3247 local RecordVal =
3248   Q "{" * SkipSpace
3249   *
3250   (
3251     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3252   ) ^ -1
3253   *
3254   (
3255     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3256   )
3257   * SkipSpace
3258   * Q ";" ^ -1
3259   * SkipSpace
3260   * Comment ^ -1
3261   * SkipSpace
3262   * Q "}"
3263 local RecordType =
3264   Q "{" * SkipSpace
3265   *
3266   (
3267     OneFieldDefinition
3268     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3269   )
3270   * SkipSpace
3271   * Q ";" ^ -1
3272   * SkipSpace
3273   * Comment ^ -1
3274   * SkipSpace
3275   * Q "}"
3276 local Record = RecordType + RecordVal

```

```

3277 local Operator =
3278   P "||" *

```

Don't use \hspace instead of \kern!

```

3279 Lc([{\PitonStyle{Operator}{\kern0.1em/\kern-0.2em/\kern0.1em}}])
3280 +
3281 K ( 'Operator' ,
3282   P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3283   "//" + "*" + ";" + "->" + "+." + "-." + "*." + "/" +
3284   + S "--+/*%=<>&@|" )

3285 local Builtin =
3286   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

```

```

3287 local Exception =
3288   K ( 'Exception' ,
3289     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3290     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3291     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

3292 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).

```

3293 local pattern_part =
3294   ( 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).

```
3295 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.

```

3296   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3297   *

```

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

```

3298   (
3299     K ( 'Identifier.Internal' , identifier )
3300     +
3301     Q "(" * SkipSpace
3302     * ( C ( pattern_part ) / ParseAgain )
3303     * SkipSpace

```

Of course, the specification of type is optional.

```

3304     * ( Q ":" * #(1- P"=")
3305         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3306     ) ^ -1
3307     * Q ")"
3308   )

```

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

```

3309 local DefFunction =
3310   K ( 'Keyword.Governing' , "let open" )
3311   * Space
3312   * K ( 'Name.Module' , cap_identifier )
3313   +
3314   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3315   * Space
3316   * K ( 'Name.Function.Internal' , identifier )
3317   * Space
3318   * (

```

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).

```

3319     Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3320     +
3321     Argument * ( SkipSpace * Argument ) ^ 0
3322     * (
3323       SkipSpace
3324       * Q ":" * #( 1 - P "=" )
3325       * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3326     ) ^ -1
3327   )

```

DefModule

```

3328 local DefModule =
3329   K ( 'Keyword.Governing' , "module" ) * Space
3330   *
3331   (
3332     K ( 'Keyword.Governing' , "type" ) * Space
3333     * K ( 'Name.Type' , cap_identifier )
3334     +
3335     K ( 'Name.Module' , cap_identifier ) * SkipSpace
3336     *
3337     (
3338       Q "(" * SkipSpace
3339       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3340       * Q ":" * # ( 1 - P "=" ) * SkipSpace
3341       * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3342       *
3343       (
3344         Q "," * SkipSpace
3345         * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3346         * Q ":" * # ( 1 - P "=" ) * SkipSpace
3347         * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3348       ) ^ 0
3349       * Q ")"
3350     ) ^ -1
3351   *
3352   (
3353     Q "=" * SkipSpace
3354     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3355     * Q "("
3356     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3357     *
3358     (
3359       Q ","
3360       *
3361       K ( 'Name.Module' , cap_identifier ) * SkipSpace
3362     ) ^ 0
3363     * Q ")"
3364   ) ^ -1
3365 )
3366 +
3367 K ( 'Keyword.Governing' , P "include" + "open" )
3368 * Space
3369 * K ( 'Name.Module' , cap_identifier )

```

DefType

```

3370 local DefType =
3371   K ( 'Keyword.Governing' , "type" )
3372   * Space
3373   * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3374   * SkipSpace
3375   * ( Q "+=" + Q "=" )
3376   * SkipSpace
3377   * (
3378     RecordType
3379     +

```

The following lines are a suggestion of Y. Salmon.

```

3380 WithStyle
3381 (
3382   'TypeExpression' ,
3383   (
3384     (
3385       EOL

```

```

3386         + comment
3387         + Q ( 1
3388             - P ";;"
3389             - P "type"
3390             - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3391         )
3392     ) ^ 0
3393     *
3394     (
3395         # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3396         + Q ";;"
3397         + -1
3398     )
3399 )
3400 )
3401 )

3402 local prompt =
3403     Q "utop[" * digit^1 * Q "> "
3404 local start_of_line = P(function(subject, position)
3405     if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3406         return position
3407     end
3408     return nil
3409 end)
3410 local Prompt = #start_of_line * K( 'Prompt', prompt )
3411 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3412     * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3413     * ( K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

The main LPEG for the language OCaml

```

3414 local Main =
3415     space ^ 0 * EOL
3416     + Space
3417     + Tab
3418     + Escape + EscapeMath
3419     + Beamer
3420     + DetectedCommands
3421     + TypeParameter
3422     + String + QuotedString + Char
3423     + Comment
3424     + Prompt + Answer

```

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

```

3425     + Q "~" * Identifier * ( Q ":" ) ^ -1
3426     + Q ":" * # (1 - P ":" ) * SkipSpace
3427     * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3428     + Exception
3429     + DefType
3430     + DefFunction
3431     + DefModule
3432     + Record
3433     + Keyword * EndKeyword
3434     + OperatorWord * EndKeyword
3435     + Builtin * EndKeyword
3436     + DotNotation * EndKeyword
3437     + Constructor
3438     + Identifier
3439     + Punct
3440     + Delim -- Delim is before Operator for a correct analysis of [| et |]
3441     + Operator

```

```

3442     + Number
3443     + Word

```

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

```

3444     LPEG1.ocaml = Main ^ 0

```

```

3445     LPEG2.ocaml =
3446     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).

```

3447         (
3448         (
3449             P ":"
3450             +
3451             (
3452                 ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3453                 * Identifier
3454                 * SkipSpace
3455                 * Q ":"
3456             )
3457         )
3458         * # ( 1 - S ":@" )
3459         * SkipSpace
3460         * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 ) * -1
3461     )
3462     +
3463     (
3464         ( space ^ 0 * "\r" ) ^ -1
3465         * Lc [[ \@@_begin_line: ]]
3466         * LeadingSpace ^ 0
3467         * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
3468           + space ^ 0 * EOL
3469           + Main
3470         ) ^ 0
3471         * -1
3472         * Lc [[ \@@_end_line: ]]
3473     )
3474 )

```

End of the Lua scope for the language OCaml.

```

3475 end

```

3.6 The language C

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

```

3476 --c C c++ C++
3477 do

3478     local Delim = Q ( S "{[()]} " )
3479     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.

```

3480     local identifier = letter * alphanum ^ 0
3481
3482     local Operator =

```

```

3483 K ( 'Operator' ,
3484     P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3485     + S "~+/*%=<>&.@|!" )
3486
3487 local Keyword =
3488     K ( 'Keyword' ,
3489         P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3490         "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3491         "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3492         "register" + "restricted" + "return" + "static" + "static_assert" +
3493         "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3494         "union" + "using" + "virtual" + "volatile" + "while"
3495     )
3496     + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3497
3498 local Builtin =
3499     K ( 'Name.Builtin' ,
3500         P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3501
3502 local Type =
3503     K ( 'Name.Type' ,
3504         P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3505         "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3506         "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3507         "void" + "wchar_t" ) * Q "*" ^ 0
3508
3509 local DefFunction =
3510     Type
3511     * Space
3512     * Q "*" ^ -1
3513     * K ( 'Name.Function.Internal' , identifier )
3514     * SkipSpace
3515     * # 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:`

```

3516 local DefClass =
3517     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).

```

3518 local Character =
3519     K ( 'String.Short' ,
3520         P "['\''"] + P "\"" * ( 1 - P "\"" ) ^ 0 * P "\"" )

```

The strings of C

```

3521 String =
3522     WithStyle ( 'String.Long.Internal' ,
3523         Q "\"\"
3524         * ( SpaceInString
3525             + K ( 'String.Interpol' ,
3526                 "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3527             )
3528             + Q ( ( P "\\\"\" + 1 - S " \"\" ) ^ 1 )
3529             ) ^ 0
3530         * Q "\"\"
3531     )

```

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

```

3532 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
3533 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end

3534 DetectedCommands =
3535   Compute_DetectedCommands ( 'c' , braces )
3536   + Compute_RawDetectedCommands ( 'c' , braces )

3537 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

The directives of the preprocessor

```

3538 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.

```

3539 local Comment =
3540   WithStyle ( 'Comment.Internal' ,
3541     Q "//" * ( CommentMath + Q ( ( 1 - S "$r" ) ^ 1 ) ) ^ 0 ) -- $
3542     * ( EOL + -1 )
3543
3544 local LongComment =
3545   WithStyle ( 'Comment.Internal' ,
3546     Q "/*"
3547     * ( CommentMath + Q ( ( 1 - P "*/" - S "$r" ) ^ 1 ) + EOL ) ^ 0
3548     * Q "*/"
3549     ) -- $

```

The main LPEG for the language C

```

3550 local EndKeyword
3551   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3552     EscapeMath + -1

```

First, the main loop :

```

3553 local Main =
3554   space ^ 0 * EOL
3555   + Space
3556   + Tab
3557   + Escape + EscapeMath
3558   + CommentLaTeX
3559   + Beamer
3560   + DetectedCommands
3561   + Preproc
3562   + Comment + LongComment
3563   + Delim
3564   + Operator
3565   + Character
3566   + String
3567   + Punct
3568   + DefFunction
3569   + DefClass
3570   + Type * ( Q "*" ^ -1 + EndKeyword )
3571   + Keyword * EndKeyword
3572   + Builtin * EndKeyword
3573   + Identifier
3574   + Number
3575   + Word

```


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

```
3576 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:`⁷.

```
3577 LPEG2.c =
3578 Ct (
3579   ( space ^ 0 * P "\r" ) ^ -1
3580   * Lc [[ \@@_begin_line: ]]
3581   * LeadingSpace ^ 0
3582   * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3583   * -1
3584   * Lc [[ \@@_end_line: ]]
3585 )
```

End of the Lua scope for the language C.

```
3586 end
```

3.7 The language SQL

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

```
3587 --sql SQL
3588 do

3589   local LuaKeyword
3590   function LuaKeyword ( name ) return
3591     Lc [[ {\PitonStyle{Keyword}{ }}
3592     * Q ( Cmt (
3593       C ( letter * alphanum ^ 0 ) ,
3594       function ( _ , _ , a ) return a : upper ( ) == name end
3595     )
3596   )
3597   * Lc "}"
3598 end
```

In the identifiers, we will be able to catch those contening spaces, that is to say like "last name".

```
3599 local identifier =
3600   letter * ( alphanum + "-" ) ^ 0
3601   + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"

3602 local Operator =
3603   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).

```
3604 local Set
3605 function Set ( list )
3606   local set = { }
3607   for _ , l in ipairs ( list ) do set[l] = true end
3608   return set
3609 end
```

⁷Remember 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.

```

3610 local set_keywords = Set
3611 {
3612     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3613     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3614     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3615     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3616     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3617     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3618     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3619     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3620     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3621     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3622     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3623     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
3624     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3625     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3626     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3627     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3628     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3629     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3630     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3631     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3632 }
3633
3634 local set_builtins = Set
3635 {
3636     "AVG", "COUNT", "CHAR_LENGTH", "CONCAT", "CURDATE", "CURRENT_DATE",
3637     "DATE_FORMAT", "DAY", "LOWER", "LTRIM", "MAX", "MIN", "MONTH", "NOW",
3638     "RANK", "ROUND", "RTRIM", "SUBSTRING", "SUM", "UPPER", "YEAR"
3639 }

```

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.

```

3639 local Identifier =
3640   C ( identifier ) /
3641   (
3642     function ( s )
3643       if set_keywords [ s : upper ( ) ] then return

```

Remind that, in Lua, it's possible to return *several* values.

```

3644     { [{\PitonStyle{Keyword}}{[] } } ,
3645     { luatexbase.catcodetables.other , s } ,
3646     { "}" } }
3647   else
3648     if set_builtins [ s : upper ( ) ] then return
3649     { [{\PitonStyle{Name.Builtin}}{[] } } ,
3650     { luatexbase.catcodetables.other , s } ,
3651     { "}" } }
3652   else return
3653   { [{\PitonStyle{Name.Field}}{[] } } ,
3654   { luatexbase.catcodetables.other , s } ,
3655   { "}" } }
3656   end
3657 end
3658 end
3659 )

```

The strings of SQL

```

3660 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.

```

3661 local braces = Compute_braces ( "'" * ( 1 - P "'" ) ^ 1 * "'" )
3662 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
3663
3664 DetectedCommands =
3665   Compute_DetectedCommands ( 'sql' , braces )
3666   + Compute_RawDetectedCommands ( 'sql' , braces )
3667
3668 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.

```

3667 local Comment =
3668   WithStyle ( 'Comment.Internal' ,
3669     Q "--" -- syntax of SQL92
3670     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3671     * ( EOL + -1 )
3672
3673 local LongComment =
3674   WithStyle ( 'Comment.Internal' ,
3675     Q "/*"
3676     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3677     * Q "*/"
3678     ) -- $

```

The main LPEG for the language SQL

```

3679 local EndKeyword
3680   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3681     EscapeMath + -1
3682
3683 local TableField =
3684   K ( 'Name.Table' , identifier )
3685   * Q "."
3686   * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
3687
3688 local OneField =
3689   (
3690     Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
3691     +
3692     K ( 'Name.Table' , identifier )
3693     * Q "."
3694     * K ( 'Name.Field' , identifier )
3695     +
3696     K ( 'Name.Field' , identifier )
3697   )
3698   * (
3699     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
3700     ) ^ -1
3701   * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
3702
3703 local OneTable =
3704   K ( 'Name.Table' , identifier )
3705   * (
3706     Space
3707     * LuaKeyword "AS"
3708     * Space
3709     * K ( 'Name.Table' , identifier )
3710     ) ^ -1
3711
3712 local WeCatchTableNames =

```

```

3712     LuaKeyword "FROM"
3713     * ( Space + EOL )
3714     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
3715     + (
3716         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
3717         + LuaKeyword "TABLE"
3718     )
3719     * ( Space + EOL ) * OneTable
3720 local EndKeyword
3721     = Space + Punct + Delim + EOL + Beamer
3722     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

3723 local Main =
3724     space ^ 0 * EOL
3725     + Space
3726     + Tab
3727     + Escape + EscapeMath
3728     + CommentLaTeX
3729     + Beamer
3730     + DetectedCommands
3731     + Comment + LongComment
3732     + Delim
3733     + Operator
3734     + String
3735     + Punct
3736     + WeCatchTableNames
3737     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
3738     + Number
3739     + Word

```

Here, we must not put local, of course.

```

3740 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:`⁸.

```

3741 LPEG2.sql =
3742     Ct (
3743         ( space ^ 0 * "\r" ) ^ -1
3744         * Lc [[ \@@_begin_line: ]]
3745         * LeadingSpace ^ 0
3746         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3747         * -1
3748         * Lc [[ \@@_end_line: ]]
3749     )

```

End of the Lua scope for the language SQL.

```

3750 end

```

3.8 The language “Minimal”

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

```

3751 --minimal Minimal
3752 do

```

⁸Remember 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:`

```

3753 local Punct = Q ( S ",:;!\" )
3754
3755 local Comment =
3756   WithStyle ( 'Comment.Internal' ,
3757     Q "\""
3758     * ( CommentMath + Q ( ( 1 - S "$\" ) ^ 1 ) ) ^ 0 -- $
3759     )
3760     * ( EOL + -1 )
3761
3762 local String =
3763   WithStyle ( 'String.Short.Internal' ,
3764     Q "\""
3765     * ( SpaceInString
3766       + Q ( ( P "[\""] + 1 - S " \" ) ^ 1 )
3767       ) ^ 0
3768     * Q "\""
3769     )

```

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

```

3770 local braces = Compute_braces ( P "\"" * ( P "\\\"" + 1 - P "\"" ) ^ 1 * "\"" )
3771
3772 if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
3773
3774 DetectedCommands =
3775   Compute_DetectedCommands ( 'minimal' , braces )
3776   + Compute_RawDetectedCommands ( 'minimal' , braces )
3777
3778 LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
3779
3780 local identifier = letter * alphanum ^ 0
3781
3782 local Identifier = K ( 'Identifier.Internal' , identifier )
3783
3784 local Delim = Q ( S "{[()]}" )
3785
3786 local Main =
3787   space ^ 0 * EOL
3788   + Space
3789   + Tab
3790   + Escape + EscapeMath
3791   + CommentLaTeX
3792   + Beamer
3793   + DetectedCommands
3794   + Comment
3795   + Delim
3796   + String
3797   + Punct
3798   + Identifier
3799   + Number
3800   + Word

```

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

```

3801 LPEG1.minimal = Main ^ 0
3802
3803 LPEG2.minimal =
3804   Ct (
3805     ( space ^ 0 * "\" ) ^ -1
3806     * Lc [ [ @@_begin_line: ] ]
3807     * LeadingSpace ^ 0
3808     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3809     * -1

```

```

3810         * Lc [[ \@@_end_line: ]]
3811     )

```

End of the Lua scope for the language “Minimal”.

```

3812 end

```

3.9 The language “Verbatim”

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

```

3813 --verbatim Verbatim
3814 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”).

```

3815     local braces =
3816         P { "E" ,
3817             E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3818         }
3819
3820     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
3821
3822     DetectedCommands =
3823         Compute_DetectedCommands ( 'verbatim' , braces )
3824         + Compute_RawDetectedCommands ( 'verbatim' , braces )
3825
3826     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

3827     local lpeg_central = 1 - S " \\r"
3828     if piton.begin_escape then
3829         lpeg_central = lpeg_central - piton.begin_escape
3830     end
3831     if piton.begin_escape_math then
3832         lpeg_central = lpeg_central - piton.begin_escape_math
3833     end
3834     local Word = Q ( lpeg_central ^ 1 )
3835
3836     local Main =
3837         space ^ 0 * EOL
3838         + Space
3839         + Tab
3840         + Escape + EscapeMath
3841         + Beamer
3842         + DetectedCommands
3843         + Q [[\]]
3844         + Word

```

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

```

3845     LPEG1.verbatim = Main ^ 0
3846
3847     LPEG2.verbatim =
3848         Ct (
3849             ( space ^ 0 * "\r" ) ^ -1
3850             * Lc [[ \@@_begin_line: ]]
3851             * LeadingSpace ^ 0
3852             * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3853             * -1
3854             * Lc [[ \@@_end_line: ]]
3855         )

```

End of the Lua scope for the language “verbatim”.

```

3856 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).

```

3857 --EXPL expl
3858 do
3859     local Comment =
3860         WithStyle
3861         ( 'Comment.Internal' ,
3862           Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3863         )
3864     * ( EOL + -1 )

```

First, we begin with a special function to analyse the “keywords”, that is to say the control sequences beginning by “\”.

```

3865     local analyze_cs
3866     function analyze_cs ( s )
3867         local i = s : find ( ":" )
3868         if i then

```

First, the case of what might be called a “function” in `expl`, for instance, `\tl_set:Nn` or `\int_compare:nNnTF`.

```

3869         local name = s : sub ( 2 , i - 1 )
3870         local parts = name : explode ( "_" )
3871         local module = parts[1]
3872         if module == "" then module = parts[3] end

```

Remind that, in Lua, we can return *several* values.

```

3873         return
3874         { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3875           { luatexbase.catcodetables.other , s } ,
3876           { "}" } }
3877     else
3878         local p = s : sub ( 1 , 3 )
3879         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`.

```

3880         local scope = s : sub(2,2)
3881         local parts = s : explode ( "_" )
3882         local module = parts[2]
3883         if module == "" then module = parts[3] end
3884         local type = parts[#parts]
3885         return
3886         { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
3887           { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3888           { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,
3889           { luatexbase.catcodetables.other , s } ,
3890           { "}}}}"} }
3891     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.

```

3892         return { luatexbase.catcodetables.other , s }
3893     end
3894 end
3895 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`).

```

3896     local braces =
3897     P { "E" ,
3898       E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3899     }

```

```

3900
3901 if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
3902
3903 DetectedCommands =
3904   Compute_DetectedCommands ( 'expl' , braces )
3905   + Compute_RawDetectedCommands ( 'expl' , braces )
3906
3907 LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
3908 local control_sequence = P "\\\" * ( R "Az" + "_" + ":" + "@" ) ^ 1
3909 local ControlSequence = C ( control_sequence ) / analyze_cs
3910
3911 local def_function
3912   = P [[\cs_]]
3913   * ( P "set" + "new" )
3914   * ( P "_protected" ) ^ -1
3915   * P ":N" * ( P "p" ) ^ -1 * "n"
3916
3917 local DefFunction =
3918   C ( def_function ) / analyze_cs
3919   * Space
3920   * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
3921   * ControlSequence -- Q ( ControlSequence ) ?
3922   * Lc "}"
3923
3924 local Word = Q ( ( 1 - S " \r" ) ^ 1 )
3925
3926 local Main =
3927   space ^ 0 * EOL
3928   + Space
3929   + Tab
3930   + Escape + EscapeMath
3931   + Beamer
3932   + Comment
3933   + DetectedCommands
3934   + DefFunction
3935   + ControlSequence
3936   + Word

```

Here, we must not put local, of course.

```

3934 LPEG1.expl = Main ^ 0
3935
3936 LPEG2.expl =
3937   Ct (
3938     ( space ^ 0 * "\r" ) ^ -1
3939     * Lc [[ \@@_begin_line: ]]
3940     * LeadingSpace ^ 0
3941     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3942     * -1
3943     * Lc [[ \@@_end_line: ]]
3944   )

```

End of the Lua scope for the language expl of LaTeX3.

```

3945 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.

```

3946 function piton.Parse ( language , code )

```


The variable `piton.language` will be used by the function `ParseAgain`.

```

3947   piton.language = language
3948   local t = LPEG2[language] : match ( code )
3949   if not t then
3950     sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
3951     return -- to exit in force the function
3952   end
3953   local left_stack = {}
3954   local right_stack = {}
3955   for _ , one_item in ipairs ( t ) do
3956     if one_item == "EOL" then
3957       for i = #right_stack, 1, -1 do
3958         tex.sprint ( right_stack[i] )
3959       end

```

We remind that the `\@@_end_line:` must be explicit since it's the marker of end of the command `\@@_begin_line:`.

```

3960     sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
3961     tex.sprint ( table.concat ( left_stack ) )
3962   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}`.

```

3963     if one_item[1] == "Open" then
3964       tex.sprint ( one_item[2] )
3965       table.insert ( left_stack , one_item[2] )
3966       table.insert ( right_stack , one_item[3] )
3967     else
3968       if one_item[1] == "Close" then
3969         tex.sprint ( right_stack[#right_stack] )
3970         left_stack[#left_stack] = nil
3971         right_stack[#right_stack] = nil
3972       else
3973         tex.tprint ( one_item )
3974       end
3975     end
3976   end
3977 end
3978 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).

```

3979 local my_file_lines
3980 function my_file_lines ( filename )
3981   local f = io.open ( filename , 'rb' )
3982   local s = f : read ( '*a' )
3983   f : close ( )

```

À la fin, on doit bien mettre `(.-)` et pas `(.*)`.

```

3984   return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
3985 end

```

Recall that, in Lua, `gmatch` returns an *iterator*.

```

3986 function piton.ReadFile ( name , first_line , last_line )
3987   local s = ''
3988   local i = 0
3989   for line in my_file_lines ( name ) do

```

```

3990     i = i + 1
3991     if i >= first_line then
3992         s = s .. '\r' .. line
3993     end
3994     if i >= last_line then break end
3995 end

```

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

```

3996     if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
3997         s = s : sub ( 5 , -1 )
3998     end
3999     sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4000     tex.sprint ( luatexbase.catcodetables.other , s )
4001     sprintL3 ( "]" )
4002 end

4003 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
4004     local s
4005     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4006     piton.GobbleParse ( lang , n , splittable , s )
4007 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 `#`.

```

4008 function piton.ParseBis ( lang , code )
4009     return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
4010 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.

```

4011 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:`.

```

4012     return piton.Parse
4013     (
4014         lang ,
4015         code : gsub ( [[\@@_breakable_space: ]] , ' ' )
4016     )
4017 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.

```

4018 local AutoGobbleLPEG =
4019     ( (
4020         P " " ^ 0 * "\r"

```

```

4021      +
4022      Ct ( C " " ^ 0 ) / table.getn
4023      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
4024    ) ^ 0
4025    * ( Ct ( C " " ^ 0 ) / table.getn
4026      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4027  ) / math.min

```

The following LPEG is similar but works with the tabulations.

```

4028 local TabsAutoGobbleLPEG =
4029   (
4030     (
4031       P "\t" ^ 0 * "\r"
4032     +
4033       Ct ( C "\t" ^ 0 ) / table.getn
4034       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
4035     ) ^ 0
4036     * ( Ct ( C "\t" ^ 0 ) / table.getn
4037       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4038   ) / 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).

```

4039 local EnvGobbleLPEG =
4040   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
4041   * 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.

```

4042 function piton.Gobble ( n , code )
4043   if n == 0 then return
4044     code
4045   else
4046     if n == -1 then
4047       n = AutoGobbleLPEG : match ( code )

```

for the case of an empty environment (only blank lines)

```

4048     if tonumber(n) then else n = 0 end
4049   else
4050     if n == -2 then
4051       n = EnvGobbleLPEG : match ( code )
4052     else
4053       if n == -3 then
4054         n = TabsAutoGobbleLPEG : match ( code )
4055         if tonumber(n) then else n = 0 end
4056       end
4057     end
4058   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...

```

4059     if n == 0 then return
4060       code
4061     else return

```

We will now use a LPEG that we have to compute dynamically because it depends on the value of n .

```

4062   ( Ct (
4063     ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4064     * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4065   ) ^ 0 )
4066   / table.concat

```

```

4067     ) : match ( code )
4068     end
4069 end
4070 end

```

In the following code, `n` is the value of `\l_@@_gobble_int`.
`splittable` is the value of `\l_@@_splittable_int`.

```

4071 function piton.GobbleParse ( lang , n , splittable , code )
4072     piton.ComputeLinesStatus ( code , splittable )
4073     piton.last_code = piton.Gobble ( n , code )
4074     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`.

```

4075     piton.CountLines ( piton.last_code )
4076     piton.Parse ( lang , piton.last_code )
4077     piton.join_and_write ( )
4078 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`.

```

4079 function piton.join_and_write ( )
4080     if piton.join ~= '' then
4081         if not piton.join_files [ piton.join ] then
4082             piton.join_files [ piton.join ] = piton.get_last_code ( )
4083         else
4084             if piton.join_separation == '' then
4085                 piton.join_files [ piton.join ] =
4086                     piton.join_files [ piton.join ]
4087                     .. "\r\n"
4088                 .. piton.get_last_code ( )
4089             else
4090                 piton.join_files [ piton.join ] =
4091                     piton.join_files [ piton.join ]
4092                     .. "\r\n"
4093                     .. ( piton.join_separation : gsub ( '##' , '#' ) )
4094                     .. "\r\n"
4095                     .. piton.get_last_code ( )
4096             end
4097         end
4098     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.

```

4099     if piton.write ~= '' then

```

We will write on `file_name` the full name (with the path) of the file in which we will write.

```

4100         local file_name = ''
4101         if piton.path_write == '' then
4102             file_name = piton.write
4103         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.

```

4104             local attr = lfs.attributes ( piton.path_write )
4105             if attr and attr.mode == "directory" then
4106                 file_name = piton.path_write .. "/" .. piton.write
4107             else

```

If the directory does *not* exist, you raise an (non-fatal) error since TeX is not able to create a new directory.

```

4108      sprintL3 [[ \@_error_or_warning:n { InexistentDirectory } ]]
4109      end
4110      end
4111      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`).

```

4112      if not piton.write_files [ file_name ] then
4113          piton.write_files [ file_name ] = piton.get_last_code ( )
4114      else
4115          piton.write_files [ file_name ] =
4116          piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4117      end
4118      end
4119      end
4120  end

```

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

```

4121 function piton.GobbleParseNoPrint ( lang , n , code )
4122     piton.last_code = piton.Gobble ( n , code )
4123     piton.last_language = lang
4124     piton.join_and_write ( )
4125 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).

```

4126 function piton.GobbleSplitParse ( lang , n , splittable , code )
4127     local chunks
4128     chunks =
4129     (
4130         Ct (
4131             (
4132                 P " " ^ 0 * "\r"
4133             +
4134                 C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4135                     - ( P " " ^ 0 * ( P "\r" + -1 ) )
4136                 ) ^ 1
4137             )
4138         ) ^ 0
4139     )
4140     ) : match ( piton.Gobble ( n , code ) )
4141     sprintL3 [[ \begingroup ]]
4142     sprintL3
4143     (
4144         [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
4145         .. "language = " .. lang .. ","
4146         .. "splittable = " .. splittable .. "}"
4147     )
4148     for k , v in pairs ( chunks ) do
4149         if k > 1 then
4150             sprintL3 ( [[ \l_@_split_separation_tl ]] )
4151         end
4152         tex.print
4153         (
4154             [[\begin{}} .. piton.env_used_by_split .. "}" .. "\r"

```

```

4155         .. v
4156         .. [[\end{}} .. piton.env_used_by_split .. "}\r"
4157     )
4158 end
4159 sprintL3 [[ \endgroup ]]
4160 end

4161 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4162     local s
4163     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4164     piton.GobbleSplitParse ( lang , n , splittable , s )
4165 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.

```

4166 piton.string_between_chunks =
4167 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4168 .. [[ \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.

```

4169 function piton.get_last_code ( )
4170     return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4171         : gsub ( '\r\n?' , '\n' )
4172 end

```

3.14 To count the number of lines

```

4173 local CountBeamerEnvironments
4174 function CountBeamerEnvironments ( code ) return
4175     (
4176         Ct (
4177             (
4178                 P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4179                 +
4180                 ( 1 - P "\r" ) ^ 0 * "\r"
4181             ) ^ 0
4182             * ( 1 - P "\r" ) ^ 0
4183             * -1
4184         ) / table.getn
4185     ) : match ( code )
4186 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.

```

4187 function piton.CountLines ( code )
4188     local count
4189     count =
4190         ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4191             *
4192             (
4193                 space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4194                 + space ^ 0
4195             ) ^ -1
4196             * -1

```

```

4197         ) / table.getn
4198     ) : match ( code )
4199     if piton.beamer then
4200         count = count - 2 * CountBeamerEnvironments ( code )
4201     end
4202     sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4203 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).

```

4204 function piton.CountNonEmptyLines ( code )
4205     local count = 0

```

The following code is not clear. We should try to replace it by use of the `string` library of Lua.

```

4206     count =
4207         ( Ct ( ( P " " ^ 0 * "\r"
4208             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4209             * ( 1 - P "\r" ) ^ 0
4210             * -1
4211             ) / table.getn
4212         ) : match ( code )
4213     count = count + 1
4214     if piton.beamer then
4215         count = count - 2 * CountBeamerEnvironments ( code )
4216     end
4217     sprintL3
4218     ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4219 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.

```

4220 function piton.ComputeRange ( s , t , file_name )
4221     local first_line = -1
4222     local count = 0
4223     local last_found = false
4224     for line in io.lines ( file_name ) do
4225         if first_line == -1 then
4226             if line : sub ( 1 , #s ) == s then
4227                 first_line = count
4228             end
4229         else
4230             if line : sub ( 1 , #t ) == t then
4231                 last_found = true
4232                 break
4233             end
4234         end
4235         count = count + 1
4236     end
4237     if first_line == -1 then
4238         sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4239     else
4240         if not last_found then
4241             sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4242         end
4243     end
4244     sprintL3 (
4245         [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 ]'
4246         .. [[ \global \l_@@_last_line_int = ]] .. count )
4247 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`.

```
4248 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).

```
4249     local lpeg_line_beamer
4250     if piton.beamer then
4251         lpeg_line_beamer =
4252             space ^ 0
4253             * P [[\begin{]] * beamerEnvironments * "]"
4254             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4255             +
4256             space ^ 0
4257             * P [[\end{]] * beamerEnvironments * "]"
4258     else
4259         lpeg_line_beamer = P ( false )
4260     end
4261     local lpeg_empty_lines =
4262         Ct (
4263             ( lpeg_line_beamer * "\r"
4264             +
4265             P " " ^ 0 * "\r" * Cc ( 0 )
4266             +
4267             ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4268             ) ^ 0
4269             *
4270             ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4271         )
4272         * -1
4273     local lpeg_all_lines =
4274         Ct (
4275             ( lpeg_line_beamer * "\r"
4276             +
4277             ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4278             ) ^ 0
4279             *
4280             ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4281         )
4282         * -1
```

We begin with the computation of `piton.empty_lines`. It will be used in conjunction with `line-numbers`.

```
4283     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`).

```

4284   local lines_status
4285   local s = splittable
4286   if splittable < 0 then s = - splittable end
4287
4287   if splittable > 0 then
4288     lines_status = lpeg_all_lines : match ( code )
4289   else

```

Here, we should try to copy `piton.empty_lines` but it's not easy.

```

4290     lines_status = lpeg_empty_lines : match ( code )
4291     for i , x in ipairs ( lines_status ) do
4292       if x == 0 then
4293         for j = 1 , s - 1 do
4294           if i + j > #lines_status then break end
4295           if lines_status[i+j] == 0 then break end
4296           lines_status[i+j] = 2
4297         end
4298         for j = 1 , s - 1 do
4299           if i - j == 1 then break end
4300           if lines_status[i-j-1] == 0 then break end
4301           lines_status[i-j-1] = 2
4302         end
4303       end
4304     end
4305   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.

```

4306   for j = 1 , s - 1 do
4307     if j > #lines_status then break end
4308     if lines_status[j] == 0 then break end
4309     lines_status[j] = 2
4310   end

```

Now, from the end of the code.

```

4311   for j = 1 , s - 1 do
4312     if #lines_status - j == 0 then break end
4313     if lines_status[#lines_status - j] == 0 then break end
4314     lines_status[#lines_status - j] = 2
4315   end

```

```

4316   piton.lines_status = lines_status
4317 end

```

```

4318 function piton.TranslateBeamerEnv ( code )
4319   local s
4320   s =
4321   (
4322     Ct (
4323       (
4324         space ^ 0
4325         * C (
4326           ( P "\\begin{" + "\\end{" )
4327           * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * "\r"
4328         )
4329         + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4330       ) ^ 0
4331     *
4332     (

```

```

4333         (
4334             space ^ 0
4335             * C (
4336                 ( P "\\begin{" + "\\end{" )
4337                 * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * -1
4338             )
4339             + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4340         ) ^ -1
4341     )
4342 ) ^ -1 / table.concat
4343 ) : match ( code )
4344 sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4345 tex.sprint ( luatexbase.catcodetables.other , s )
4346 sprintL3 ( "]" )
4347 end

```

3.16 To create new languages with the syntax of listings

```

4348 function piton.new_language ( lang , definition )
4349     lang = lang : lower ( )

4350     local alpha , digit = lpeg.alpha , lpeg.digit
4351     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`.

```

4352     function add_to_letter ( c )
4353         if c ~= " " then table.insert ( extra_letters , c ) end
4354     end

```

For the digits, it's straitforward.

```

4355     function add_to_digit ( c )
4356         if c ~= " " then digit = digit + c end
4357     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 `}`).

```

4358     local other = S " :_@+~*/<>!?.() [] ~^=#&\"'\\$" --
4359     local extra_others = { }
4360     function add_to_other ( c )
4361         if c ~= " " then

```

We will use `extra_others` to retrieve further these characters from the list of the letters.

```

4362         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 `</...>`.

```

4363         other = other + P ( c )
4364     end
4365 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`.

```

4366     local def_table
4367     if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4368         def_table = {}
4369     else
4370         local strict_braces =

```

```

4371     P { "E" ,
4372         E = ( "{" * V "F" * "}" + ( 1 - S ",{" }" ) ) ^ 0 ,
4373         F = ( "{" * V "F" * "}" + ( 1 - S "{" }" ) ) ^ 0
4374     }
4375     local cut_definition =
4376     P { "E" ,
4377         E = Ct ( V "F" * ( "," * V "F" ) ^ 0 ) ,
4378         F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4379             * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4380     }
4381     def_table = cut_definition : match ( definition )
4382 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.).

```

4383     local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4384     local tex_arg = tex_braced_arg + C ( 1 )
4385     local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4386     local args_for_tag
4387     = tex_option_arg
4388     * space ^ 0
4389     * tex_arg
4390     * space ^ 0
4391     * tex_arg
4392     local args_for_morekeywords
4393     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4394     * space ^ 0
4395     * tex_option_arg
4396     * space ^ 0
4397     * tex_arg
4398     * space ^ 0
4399     * ( tex_braced_arg + Cc ( nil ) )
4400     local args_for_moredelims
4401     = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4402     * args_for_morekeywords
4403     local args_for_morecomment
4404     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4405     * space ^ 0
4406     * tex_option_arg
4407     * space ^ 0
4408     * 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`.

```

4409     local sensitive = true
4410     local style_tag , left_tag , right_tag
4411     for _ , x in ipairs ( def_table ) do
4412         if x[1] == "sensitive" then
4413             if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4414                 sensitive = true
4415             else
4416                 if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4417             end
4418         end
4419         if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4420         if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4421         if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4422         if x[1] == "tag" then

```

```

4423     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4424     style_tag = style_tag or [[\PitonStyle{Tag}]]
4425 end
4426 end

```

Now, the LPEG for the numbers. Of course, it uses `digit` previously computed.

```

4427 local Number =
4428   K ( 'Number.Internal' ,
4429     ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4430       + digit ^ 0 * "." * digit ^ 1
4431       + digit ^ 1 )
4432     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4433     + digit ^ 1
4434   )
4435
4436 local string_extra_letters = ""
4437 for _ , x in ipairs ( extra_letters ) do
4438   if not ( extra_others[x] ) then
4439     string_extra_letters = string_extra_letters .. x
4440   end
4441 end
4442 local letter = alpha + S ( string_extra_letters )
4443   + P "â" + "à" + "ç" + "ê" + "ë" + "ê" + "ï" + "î"
4444   + "ô" + "û" + "ü" + "Ë" + "Ê" + "Ç" + "É" + "È" + "Ê" + "Ë"
4445   + "İ" + "Ĭ" + "Ō" + "Ū" + "Ū"
4446
4447 local alphanum = letter + digit
4448 local identifier = letter * alphanum ^ 0
4449 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.

```

4448 local split_clist =
4449   P { "E" ,
4450     E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4451     * ( P "{" ) ^ 1
4452     * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4453     * ( P "}" ) ^ 1 * space ^ 0 ,
4454     F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4455   }

```

The following function will be used if the keywords are not case-sensitive.

```

4456 local keyword_to_lpeg
4457 function keyword_to_lpeg ( name ) return
4458   Q ( Cmt (
4459     C ( identifier ) ,
4460     function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4461   end
4462 ) )
4463 end
4464
4465 local Keyword = P ( false )
4466 local PrefixedKeyword = P ( false )

```

Now, we actually treat all the keywords and also the key `moredirectives`.

```

4467 for _ , x in ipairs ( def_table )
4468 do if x[1] == "morekeywords"
4469   or x[1] == "otherkeywords"
4470   or x[1] == "moredirectives"
4471   or x[1] == "moretexcs"
4472 then
4473   local keywords = P ( false )
4474   local style = [[\PitonStyle{Keyword}]]
4475   if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
4476   style = tex_option_arg : match ( x[2] ) or style
4477   local n = tonumber ( style )

```

```

4478     if n then
4479         if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4480     end
4481     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4482         if x[1] == "moretexcs" then
4483             keywords = Q ( [[\]] .. word ) + keywords
4484         else
4485             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*.

```

4486         then keywords = Q ( word ) + keywords
4487         else keywords = keyword_to_lpeg ( word ) + keywords
4488     end
4489 end
4490 end
4491 Keyword = Keyword +
4492     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4493 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.

```

4494     if x[1] == "keywordsprefix" then
4495         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4496         PrefixedKeyword = PrefixedKeyword
4497             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4498     end
4499 end

```

Now, we scan the definition of the language (i.e. the table `def_table`) for the strings.

```

4500     local long_string = P ( false )
4501     local Long_string = P ( false )
4502     local LongString = P ( false )
4503     local central_pattern = P ( false )
4504     for _ , x in ipairs ( def_table ) do
4505         if x[1] == "morestring" then
4506             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4507             arg2 = arg2 or [[\PitonStyle{String.Long}]]
4508             if arg1 ~= "s" then
4509                 arg4 = arg3
4510             end
4511             central_pattern = 1 - S ( " \r" .. arg4 )
4512             if arg1 : match "b" then
4513                 central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4514             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...

```

4515         if arg1 : match "d" or arg1 == "m" then
4516             central_pattern = P ( arg3 .. arg3 ) + central_pattern
4517         end
4518         if arg1 == "m"
4519         then prefix = B ( 1 - letter - " ) - "]" )
4520         else prefix = P ( true )
4521         end

```

First, a pattern *without captures* (needed to compute braces).

```

4522     long_string = long_string +
4523         prefix
4524         * arg3
4525         * ( space + central_pattern ) ^ 0
4526         * arg4

```

Now a pattern *with captures*.

```

4527     local pattern =
4528         prefix
4529         * Q ( arg3 )
4530         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4531         * Q ( arg4 )

```

We will need Long_string in the nested comments.

```

4532     Long_string = Long_string + pattern
4533     LongString = LongString +
4534         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4535         * pattern
4536         * Ct ( Cc "Close" )
4537     end
4538 end

```

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

```

4539     local braces = Compute_braces ( long_string )
4540     if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4541
4542     DetectedCommands =
4543         Compute_DetectedCommands ( lang , braces )
4544         + Compute_RawDetectedCommands ( lang , braces )
4545
4546     LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )

```

Now, we deal with the comments and the delims.

```

4547     local CommentDelim = P ( false )
4548
4549     for _ , x in ipairs ( def_table ) do
4550         if x[1] == "morecomment" then
4551             local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4552             arg2 = arg2 or [[\PitonStyle{Comment}]]

```

If the letter i is present in the first argument (eg: morecomment = [si]{(*){*}), then the corresponding comments are discarded.

```

4553             if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4554             if arg1 : match "l" then
4555                 local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4556                     : match ( other_args )
4557                 if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4558                 if arg3 == [[\%]] then arg3 = "%" end -- mandatory
4559                 CommentDelim = CommentDelim +
4560                     Ct ( Cc "Open"
4561                         * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4562                         * Q ( arg3 )
4563                         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4564                         * Ct ( Cc "Close" )
4565                         * ( EOL + -1 )
4566             else
4567                 local arg3 , arg4 =
4568                     ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
4569                 if arg1 : match "s" then
4570                     CommentDelim = CommentDelim +
4571                         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4572                         * Q ( arg3 )

```

```

4573         * (
4574             CommentMath
4575             + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
4576             + EOL
4577         ) ^ 0
4578         * Q ( arg4 )
4579         * Ct ( Cc "Close" )
4580     end
4581     if arg1 : match "n" then
4582         CommentDelim = CommentDelim +
4583         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4584         * P { "A" ,
4585             A = Q ( arg3 )
4586             * ( V "A"
4587                 + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
4588                     - S "\r$" ) ^ 1 ) -- $
4589                 + long_string
4590                 + "$" -- $
4591                 * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
4592                 * "$" -- $
4593                 + EOL
4594             ) ^ 0
4595             * Q ( arg4 )
4596         }
4597         * Ct ( Cc "Close" )
4598     end
4599 end
4600 end

```

For the `moredelim`, we have to add another argument in first position, equal to `*` or `**`.

```

4601 if x[1] == "moredelim" then
4602     local arg1 , arg2 , arg3 , arg4 , arg5
4603     = args_for_moredelims : match ( x[2] )
4604     local MyFun = Q
4605     if arg1 == "*" or arg1 == "**" then
4606         function MyFun ( x )
4607             if x ~= '' then return
4608                 LPEG1[lang] : match ( x )
4609             end
4610         end
4611     end
4612     local left_delim
4613     if arg2 : match "i" then
4614         left_delim = P ( arg4 )
4615     else
4616         left_delim = Q ( arg4 )
4617     end
4618     if arg2 : match "l" then
4619         CommentDelim = CommentDelim +
4620         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4621         * left_delim
4622         * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4623         * Ct ( Cc "Close" )
4624         * ( EOL + -1 )
4625     end
4626     if arg2 : match "s" then
4627         local right_delim
4628         if arg2 : match "i" then
4629             right_delim = P ( arg5 )
4630         else
4631             right_delim = Q ( arg5 )
4632         end
4633         CommentDelim = CommentDelim +
4634         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )

```

```

4635         * left_delim
4636         * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
4637         * right_delim
4638         * Ct ( Cc "Close" )
4639     end
4640 end
4641 end
4642
4643 local Delim = Q ( S "{[()]}" )
4644 local Punct = Q ( S "=:;!\\"'" )
4645
4646 local Main =
4647     space ^ 0 * EOL
4648     + Space
4649     + Tab
4650     + Escape + EscapeMath
4651     + CommentLaTeX
4652     + Beamer
4653     + DetectedCommands
4654     + CommentDelim

```

We must put `LongString` before `Delim` because, in PostScript, the strings are delimited by parenthesis and those parenthesis would be caught by `Delim`.

```

4654     + LongString
4655     + Delim
4656     + PrefixedKeyword
4657     + Keyword * ( -1 + # ( 1 - alphanum ) )
4658     + Punct
4659     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4660     + Number
4661     + 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.

```

4662 LPEG1[lang] = Main ^ 0

```

The LPEG `LPEG2[lang]` is used to format general chunks of code.

```

4663 LPEG2[lang] =
4664     Ct (
4665         ( space ^ 0 * P "\r" ) ^ -1
4666         * Lc [[ \@@_begin_line: ]]
4667         * LeadingSpace ^ 0
4668         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
4669         * -1
4670         * Lc [[ \@@_end_line: ]]
4671     )

```

If the key `tag` has been used. Of course, this feature is designed for the languages such as HTML and XML.

```

4672 if left_tag then
4673     local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
4674     * Q ( left_tag * other ^ 0 ) -- $
4675     * ( ( 1 - P ( right_tag ) ) ^ 0 )
4676     / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
4677     * Q ( right_tag )
4678     * Ct ( Cc "Close" )
4679
4680 MainWithoutTag
4681     = space ^ 1 * -1
4682     + space ^ 0 * EOL
4683     + Space
4684     + Tab
4685     + Escape + EscapeMath
4686     + CommentLaTeX
4687     + Beamer

```



```

4687         + DetectedCommands
4688         + CommentDelim
4689         + Delim
4690         + LongString
4691         + PrefixedKeyword
4692         + Keyword * ( -1 + # ( 1 - alphanum ) )
4693         + Punct
4694         + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4695         + Number
4696         + Word
4697 LPEG0[lang] = MainWithoutTag ^ 0
4698 local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
4699               + Beamer + DetectedCommands + CommentDelim + Tag
4700 MainWithTag
4701     = space ^ 1 * -1
4702     + space ^ 0 * EOL
4703     + Space
4704     + LPEGaux
4705     + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
4706 LPEG1[lang] = MainWithTag ^ 0
4707 LPEG2[lang] =
4708     Ct (
4709         ( space ^ 0 * P "\r" ) ^ -1
4710         * Lc [[ \@_begin_line: ]]
4711         * Beamer
4712         * LeadingSpace ^ 0
4713         * LPEG1[lang]
4714         * -1
4715         * Lc [[ \@_end_line: ]]
4716     )
4717 end
4718 end

```

3.17 We write the files (key 'write') and join the files in the PDF (key 'join')

```

4719 function piton.write_files_now ( )
4720     for file_name , file_content in pairs ( piton.write_files ) do
4721         local file = io.open ( file_name , "w" )
4722         if file then
4723             file : write ( file_content )
4724             file : close ( )
4725         else
4726             sprintL3
4727             ( [[ \@_error_or_warning:nn { FileError } { ]] .. file_name .. "]" )
4728         end
4729     end
4730 end

```

3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

4731 function piton.utf16 ( str )
4732     local hex = { "FEFF" } -- BOM UTF-16BE
4733     for _, codepoint in utf8.codes(str) do
4734         table.insert(hex, string.format("%04X", codepoint))
4735     end
4736     return table.concat(hex)
4737 end
4738 </LUA>

```