

The package `piton`*

F. Pantigny
fpantigny@wanadoo.fr

July 1, 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 4.14 of `piton`, at the date of 2026/07/01.

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 the L3 Programming Layer (as set by `\ExplSyntaxOn`).

```

\_piton_begin_line:{\PitonStyle{Keyword}{def}}
\_piton_end_line:{\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      "File->~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  }
66  \sys_if_engine_luatex:F { \@@_fatal:n { LuaLaTeX-mandatory } }

67  \RequirePackage { luacode }

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

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

```

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

```

79  \keys_define:nn { piton }
80    {
81      footnote .bool_gset:N = \g_@@_footnote_bool ,
82      footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
83      footnote .usage:n = load ,
84      footnotehyper .usage:n = load ,
85
86      beamer .bool_gset:N = \g_@@_beamer_bool ,
87      beamer .usage:n = load ,
88
89      unknown .code:n = \@@_error:n { Unknown-key-for-package }
90    }
91  \@@_msg_new:nn { Unknown-key-for-package }
92    {
93      Unknown-key.\
94      You~have~used~the~key~'\l_keys_key_str'~when~loading~piton~
95      but~the~only~keys~available~here~are~'beamer',~'footnote'~
96      and~'footnotehyper'.~Other~keys~are~available~in~
97      \token_to_str:N \PitonOptions.\
98      That~key~will~be~ignored.
99    }

```

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

```

100  \ProcessKeyOptions

```

```

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

114 \RequirePackage { xcolor }

115 \@@_msg_new:nn { footnote-with-footnotehyper-package }
116 {
117     Footnote~forbidden.\
118     You~can't~use~the~option~'footnote'~because~the~package~
119     footnotehyper~has~already~been~loaded.~
120     If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
121     within~the~environments~of~piton~will~be~extracted~with~the~tools~
122     of~the~package~footnotehyper.\
123     If~you~go~on,~the~package~footnote~won't~be~loaded.
124 }

125 \@@_msg_new:nn { footnotehyper-with-footnote-package }
126 {
127     You~can't~use~the~option~'footnotehyper'~because~the~package~
128     footnote~has~already~been~loaded.~
129     If~you~want,~you~can~use~the~option~'footnote'~and~the~footnotes~
130     within~the~environments~of~piton~will~be~extracted~with~the~tools~
131     of~the~package~footnote.\
132     If~you~go~on,~the~package~footnotehyper~won't~be~loaded.
133 }

134 \bool_if:NT \g_@@_footnote_bool
135 {

```

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

```

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

144 \bool_if:NT \g_@@_footnotehyper_bool
145 {

```

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

```

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

```

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

```
155 \dim_new:N \l_@@_rounded_corners_dim
```

```
156 \bool_new:N \l_@@_in_label_bool
```

```
157 \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`).

```
158 \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:`).

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

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

```
160 \str_new:N \l_piton_language_str
```

```
161 \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.

```
162 \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`).

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

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

```
164 \seq_new:N \g_@@_join_seq
```

```
165 \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.

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

The following parameter corresponds to the key `box`.

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

In order to have a better control over the keys.

```
168 \bool_new:N \l_@@_in_PitonOptions_bool
```

```
169 \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.

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

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

```
171 \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).

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

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

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

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

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

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

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

```
176 \str_new:N \l_@@_begin_range_str
177 \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).

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

The argument of `\PitonInputFile`.

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

The following line can't be deleted.

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

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

```
181 \bool_new:N \l_@@_paperclip_bool
182 \str_new:N \l_@@_paperclip_str
```

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

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

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

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

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

```
185 \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).

```
186 \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.

```
187 \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).

```
188 \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:`

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

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

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

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

```
191 \dim_new:N \l_@@_left_margin_dim
192 \dim_new:N \l_@@_right_margin_dim
```

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

```
193 \bool_new:N \l_@@_left_margin_auto_bool
194 \bool_new:N \l_@@_right_margin_auto_bool
```

The following boolean corresponds to the key `line-numbers/lmmono10-draw`.

```
195 \bool_new:N \l_@@_lmodern_drawn_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.

```
196 \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.

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

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

```
211 \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.

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

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

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



```

228     }
229     \@esphack
230     \IfPackageLoadedT { hyperref }
231     { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
232 }
233 { \@@_error:n { label-with-lines-numbers } }
234 }

```

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`.

```

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

```

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

```

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

```

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).

```

263 \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.

```

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

```

266 \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:`.

```

267 \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`.

```

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

270 \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.

```

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

```

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

```

The L3 Programming Layer 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.

```

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

```

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.

```

290 \tl_new:N \g_@@_def_vertical_commands_tl

```

The following commands must *not* be protected.

```

291 \cs_new:Npn \@@_retrieve_backslash:n #1
292 {
293   \str_if_eq:eeTF \c_backslash_str { \str_head:n { #1 } }
294   { \str_tail:n { #1 } }
295   { #1 }
296 }

297 \cs_new_protected:Npn \@@_detected_commands:n #1
298 {
299   \clist_set:Nn \l_tmpa_clist { #1 }
300   \clist_clear:N \l_tmpb_clist
301   \clist_map_inline:Nn \l_tmpa_clist
302   { \clist_put_right:Ne \l_tmpb_clist { \@@_retrieve_backslash:n { ##1 } } }
303   \clist_if_in:NnTF \l_tmpb_clist { rowcolor }
304   {
305     \@@_error:n { rowcolor~in~detected~commands }
306     \clist_remove_all:Nn \l_tmpb_clist { rowcolor }
307   }
308   \clist_put_right:No \l_@@_detected_commands_clist \l_tmpb_clist
309 }

310 \cs_new_protected:Npn \@@_raw_detected_commands:n #1
311 {
312   \clist_set:Nn \l_tmpa_clist { #1 }
313   \clist_clear:N \l_tmpb_clist
314   \clist_map_inline:Nn \l_tmpa_clist
315   { \clist_put_right:Ne \l_tmpb_clist { \@@_retrieve_backslash:n { ##1 } } }
316   \clist_put_right:No \l_@@_raw_detected_commands_clist \l_tmpb_clist
317 }

318 \cs_new_protected:Npn \@@_vertical_detected_commands:n #1
319 {
320   \clist_set:Nn \l_tmpa_clist { #1 }
321   \clist_clear:N \l_tmpb_clist
322   \clist_map_inline:Nn \l_tmpa_clist
323   { \clist_put_right:Ne \l_tmpb_clist { \@@_retrieve_backslash:n { ##1 } } }
324   \clist_put_right:No \l_@@_raw_detected_commands_clist \l_tmpb_clist
325   \clist_map_inline:Nn \l_tmpb_clist
326   {
327     \cs_set_eq:cc { @@_old_ ##1 : } { ##1 }
328     \cs_new_protected:cn { @@_new_ ##1 : n }
329     {
330       \bool_if:nTF
331       { \l_@@_tcolorbox_bool || ! \str_if_empty_p:N \l_@@_box_str }
332       {
333         \tl_gput_right:Nn \g_@@_after_line_tl
334         { \use:c { @@_old_ ##1 : } { #####1 } }
335       }
336       {
337         \cs_if_exist:cTF { g_@@_after_line _ \int_use:N \g_@@_line_int }
338         { \tl_gput_right:cn }
339         { \tl_gset:cn }
340         { g_@@_after_line _ \int_eval:n { \g_@@_line_int + 1 }_tl }
341         { \use:c { @@_old_ ##1 : } { #####1 } }
342       }
343     }
344     \tl_gput_right:Nn \g_@@_def_vertical_commands_tl
345     { \cs_set_eq:cc { ##1 } { @@_new_ ##1 : n } }
346   }
347 }

```

2.4 Treatment of a line of code

```

348 \cs_new_protected:Npn \@@_replace_spaces:n #1
349 {
350   \tl_set:Nn \l_tmpa_tl { #1 }
351   \bool_if:NTF \l_@@_show_spaces_bool
352   {
353     \tl_set:Nn \l_@@_space_in_string_tl { } % U+2423
354     \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { } % U+2423
355   }
356   {

```

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.

```

357   \bool_if:NT \l_@@_break_lines_in_Piton_bool
358   {
359     \tl_if_eq:NnF \l_@@_space_in_string_tl { }
360     { \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 job for the *doc strings* of Python and for the comments.

```

361   \tl_replace_all:Nvn \l_tmpa_tl
362   \c_catcode_other_space_tl
363   \@@_breakable_space:
364 }
365 }
366 \l_tmpa_tl
367 }
368 \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:`).

```

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

370 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
371 {
372   \group_begin:
373   \int_gzero:N \g_@@_indentation_int

```

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

```

374   \hbox_set:Nn \l_@@_line_box
375   {
376     \skip_horizontal:N \l_@@_left_margin_dim
377     \bool_if:NT \l_@@_line_numbers_bool
378     {

```

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

```

379 \int_set:Nn \l_tmpa_int
380 {
381   \lua_now:e
382   {
383     tex.sprint
384     (

```

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`.

```

385       piton.empty_lines
386       [ \int_eval:n { \g_@@_line_int + 1 } ]
387     )
388   }
389 }
390 \bool_lazy_or:nnT
391 { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
392 { ! \l_@@_skip_empty_lines_bool }
393 { \int_gincr:N \g_@@_visual_line_int }
394
395 \bool_lazy_or:nnTF
396 { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
397 { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
398 {
399   \bool_lazy_or:nnTF
400   { \int_compare_p:nNn { \l_@@_numbers_step_int } = 1 }
401   {
402     \int_compare_p:nNn
403     {
404       \int_mod:nn
405       { \g_@@_visual_line_int }
406       { \l_@@_numbers_step_int }
407     }
408     = \c_one_int
409   }
410   {
411     \str_if_eq:eeTF \l_@@_line_numbers_position_str { left }
412     \@@_print_number_left:
413     {
414       \seq_gput_right:Ne \g_@@_visual_line_numbers_seq
415       { \int_use:N \g_@@_visual_line_int }
416     }
417   }
418   {
419     \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
420     { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
421   }
422 }
423 {
424   \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
425   { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
426 }
427 }

```

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).

```

428 \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
429 {
... but if only if the key left-margin is not used !
430   \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
431   { \skip_horizontal:n { 0.5 em } }
432 }

433 \bool_if:NTF \l_@@_minimize_width_bool
434 {

```

```

435         \hbox_set:Nn \l_tmpa_box
436         {
437             \language = -1
438             \raggedright
439             \strut
440             \@@_replace_spaces:n { #1 }
441             \strut \hfil
442         }
443         \dim_compare:nNnTF { \box_wd:N \l_tmpa_box } < \l_@@_code_width_dim
444         { \box_use:N \l_tmpa_box }
445         { \@@_vtop_of_code:n { #1 } }
446     }
447     { \@@_vtop_of_code:n { #1 } }
448 }

```

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

```

449     \box_set_dp:Nn \l_@@_line_box { \box_dp:N \l_@@_line_box + 1.25 pt }
450     \box_set_ht:Nn \l_@@_line_box { \box_ht:N \l_@@_line_box + 1.25 pt }
451     \box_use_drop:N \l_@@_line_box
452     \group_end:
453     \g_@@_after_line_tl
454     \tl_gclear:N \g_@@_after_line_tl
455 }

```

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

```

456 \cs_new_protected:Npn \@@_vtop_of_code:n #1
457 {
458     \vbox_top:n
459     {
460         \hsize = \l_@@_code_width_dim
461         \language = -1
462         \raggedright
463         \strut
464         \@@_replace_spaces:n { #1 }
465         \strut \hfil
466     }
467 }

```

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:.`

```

468 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_line_and_use:
469 {
470     \vtop
471     {
472         \offinterlineskip
473         \hbox
474         {

```

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`.

```

475         \group_begin:
476         \@@_compute_and_set_color:

```

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

```

477         \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
478         \bool_if:NT \g_@@_next_color_is_none_bool
479         { \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.

```

480     \bool_if:NTF \g_@@_color_is_none_bool
481     { \dim_zero:N \l_tmpb_dim }
482     { \dim_set_eq:NN \l_tmpb_dim \l_@@_listing_width_dim }
483     \dim_set:Nn \l_@@_tmpc_dim { \box_ht:N \l_@@_line_box }

```

Now, the colored panel.

```

484     \dim_compare:nNnTF \l_@@_rounded_corners_dim > \c_zero_dim
485     {
486         \int_compare:nNnTF \g_@@_line_int = \c_one_int
487         {
488             \begin{tikzpicture}[baseline = 0cm]
489             \fill (0,0)
490                 [rounded-corners = \l_@@_rounded_corners_dim]
491                 -- (0,\l_@@_tmpc_dim)
492                 -- (\l_tmpb_dim,\l_@@_tmpc_dim)
493                 [sharp-corners] -- (\l_tmpb_dim,-\l_tmpa_dim)
494                 -- (0,-\l_tmpa_dim)
495                 -- cycle ;
496             \end{tikzpicture}
497         }
498         {
499             \int_compare:nNnTF \g_@@_line_int = \g_@@_nb_lines_int
500             {
501                 \begin{tikzpicture}[baseline = 0cm]
502                 \fill (0,0) -- (0,\l_@@_tmpc_dim)
503                     -- (\l_tmpb_dim,\l_@@_tmpc_dim)
504                     [rounded-corners = \l_@@_rounded_corners_dim]
505                     -- (\l_tmpb_dim,-\l_tmpa_dim)
506                     -- (0,-\l_tmpa_dim)
507                     -- cycle ;
508                 \end{tikzpicture}
509             }
510             {
511                 \vrule height \l_@@_tmpc_dim
512                 depth \l_tmpa_dim
513                 width \l_tmpb_dim

```

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

```

514         \dim_compare:nNnT \l_tmpb_dim = \c_zero_dim
515         { \skip_horizontal:N \l_@@_listing_width_dim }
516     }
517 }
518 }
519 {
520     \vrule height \l_@@_tmpc_dim
521     depth \l_tmpa_dim
522     width \l_tmpb_dim

```

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

```

523     \dim_compare:nNnT \l_tmpb_dim = \c_zero_dim
524     { \skip_horizontal:N \l_@@_listing_width_dim }
525 }

```

The group is for the color of the background.

```

526     \group_end:
527     \bool_if:NT \l_@@_line_numbers_bool
528     {
529         \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
530         {
531             \seq_gpop_right:NN \g_@@_visual_line_numbers_seq \l_tmpa_tl
532             \@@_print_number_right:
533         }
534     }
535 }

```

```

536     \bool_if:NT \g_@@_next_color_is_none_bool
537     { \skip_vertical:n { 2.5 pt } }
538     \skip_vertical:n { - \box_ht_plus_dp:N \l_@@_line_box }
539     \box_use_drop:N \l_@@_line_box
540   }
541 }

```

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.

```

542 \cs_set_protected:Npn \@@_compute_and_set_color:
543 {
544   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
545   { \tl_set:Nn \l_tmpa_tl { none } }
546   {
547     \int_set:Nn \l_tmpb_int
548     { \int_mod:nn \g_@@_line_int \l_@@_bg_colors_int + 1 }
549     \tl_set:Nc \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
550   }

```

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

```

551   \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
552   {
553     \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).

```

554     \cs_undefine:c { g_@@_color_ \int_use:N \g_@@_line_int _ tl}
555   }
556   \tl_if_eq:NnTF \l_tmpa_tl { none }
557   { \bool_gset_true:N \g_@@_color_is_none_bool }
558   {
559     \bool_gset_false:N \g_@@_color_is_none_bool
560     \@@_color:o \l_tmpa_tl
561   }

```

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).

```

562   \int_compare:nNnTF { \g_@@_line_int + 1 } = \g_@@_nb_lines_int
563   { \bool_gset_false:N \g_@@_next_color_is_none_bool }
564   {
565     \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
566     { \tl_set:Nn \l_tmpa_tl { none } }
567     {
568       \int_set:Nn \l_tmpb_int
569       { \int_mod:nn { \g_@@_line_int + 1 } \l_@@_bg_colors_int + 1 }
570       \tl_set:Nc \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
571     }
572     \cs_if_exist:cT { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
573     {
574       \tl_set_eq:Nc \l_tmpa_tl
575       { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
576     }
577     \tl_if_eq:NnTF \l_tmpa_tl { none }
578     { \bool_gset_true:N \g_@@_next_color_is_none_bool }
579     { \bool_gset_false:N \g_@@_next_color_is_none_bool }
580   }
581 }

```

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} }.

```

582 \cs_set_protected:Npn \@@_color:n #1
583 {

```



```

584 \tl_if_head_eq_meaning:nNTF { #1 } [
585   {
586     \tl_set:Nn \l_tmpa_tl { #1 }
587     \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
588     \exp_last_unbraced:No \color \l_tmpa_tl
589   }
590   { \color { #1 } }
591 }
592 \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.

```

593 \cs_new_protected:Npn \@@_par:
594   {

```

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

```

595   \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.

```

596   \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.

```

597   \kern -2.5 pt

```

Now, we control page breaks after the paragraph.

```

598   \@@_add_penalty_for_the_line:
599 }

```

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`.

```

600 \cs_set_protected:Npn \@@_breakable_space:
601   {
602     \discretionary
603       { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
604       {
605         \hbox_overlap_left:n
606           {
607             {
608               \normalfont \footnotesize \color { gray }
609               \l_@@_continuation_symbol_tl
610             }
611             \skip_horizontal:n { 0.3 em }
612             \int_compare:nNnT \l_@@_bg_colors_int > \c_zero_int
613               { \skip_horizontal:n { 0.5 em } }
614           }
615         \bool_if:NT \l_@@_indent_broken_lines_bool
616         {
617           \hbox:n
618             {
619               \prg_replicate:nn { \g_@@_indentation_int } { ~ }
620               { \color { gray } \l_@@_csoi_tl }

```

```

621     }
622   }
623 }
624 { \hbox { ~ } }
625 }

```

2.5 PitonOptions

```

626 \bool_new:N \l_@@_line_numbers_bool
627 \bool_new:N \l_@@_skip_empty_lines_bool
628 \bool_set_true:N \l_@@_skip_empty_lines_bool
629 \bool_new:N \l_@@_line_numbers_absolute_bool
630 \tl_new:N \l_@@_line_numbers_format_tl
631 \tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
632 \bool_new:N \l_@@_label_empty_lines_bool
633 \bool_set_true:N \l_@@_label_empty_lines_bool
634 \int_new:N \l_@@_number_lines_start_int
635 \str_new:N \l_@@_line_numbers_position_str
636 \str_set:Nn \l_@@_line_numbers_position_str { left }
637 \bool_new:N \l_@@_resume_bool
638 \bool_new:N \g_@@_label_as_zlabel_bool
639 \tl_new:N \l_@@_after_begin_escape_tl
640 \tl_new:N \l_@@_before_end_escape_tl

641 \keys_define:nn { PitonOptions / marker }
642 {
643   beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
644   beginning .value_required:n = true ,
645   end .cs_set:Np = \@@_marker_end:n #1 ,
646   end .value_required:n = true ,
647   include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
648   unknown .code:n = \@@_error:n { Unknown-key-for-marker }
649 }

650 \keys_define:nn { PitonOptions / line-numbers }
651 {
652   true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
653   false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
654
655   start .code:n =
656     \bool_set_true:N \l_@@_line_numbers_bool
657     \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
658   start .value_required:n = true ,
659
660   skip-empty-lines .code:n =
661     \bool_if:NF \l_@@_in_PitonOptions_bool
662     { \bool_set_true:N \l_@@_line_numbers_bool }
663     \str_if_eq:nnTF { #1 } { false }
664     { \bool_set_false:N \l_@@_skip_empty_lines_bool }
665     { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
666
667   label-empty-lines .code:n =
668     \bool_if:NF \l_@@_in_PitonOptions_bool
669     { \bool_set_true:N \l_@@_line_numbers_bool }
670     \str_if_eq:nnTF { #1 } { false }
671     { \bool_set_false:N \l_@@_label_empty_lines_bool }
672     { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
673
674   absolute .code:n =
675     \bool_if:NTF \l_@@_in_PitonOptions_bool
676     { \bool_set_true:N \l_@@_line_numbers_absolute_bool }

```

```

677     { \bool_set_true:N \l_@@_line_numbers_bool }
678     \bool_if:NT \l_@@_in_PitonInputFile_bool
679     {
680         \bool_set_true:N \l_@@_line_numbers_absolute_bool
681         \bool_set_false:N \l_@@_skip_empty_lines_bool
682     } ,
683     absolute .value_forbidden:n = true ,
684
685     resume .code:n =
686         \bool_set_true:N \l_@@_resume_bool
687         \bool_if:NF \l_@@_in_PitonOptions_bool
688         { \bool_set_true:N \l_@@_line_numbers_bool } ,
689     resume .value_forbidden:n = true ,
690
691     sep .dim_set:N = \l_@@_numbers_sep_dim ,
692     sep .value_required:n = true ,
693     sep .initial:n = 0.7 em ,
694
695     step .int_set:N = \l_@@_numbers_step_int ,
696     step .initial:n = 1 ,
697     step .value_required:n = true ,
698
699     position .choices:nn = { left , right }
700     { \str_set_eq:NN \l_@@_line_numbers_position_str \l_keys_choice_tl } ,
701     position .value_required:n = true ,
702
703     format .tl_set:N = \l_@@_line_numbers_format_tl ,
704     format .value_required:n = true ,
705
706     format+ .code:n = \tl_put_right:Nn \l_@@_line_numbers_format_tl { #1 } ,
707     format+ .value_required:n = true ,
708
709     format~+ .meta:n = { format+ = #1 } ,
710
711     lmmono10-drawn .bool_set:N = \l_@@_lmodern_drawn_bool ,
712     lmmono10-drawn .default:n = true ,
713
714     unknown .code:n =
715         \@@_unknown_key:nn
716         { PitonOptions / line-numbers }
717         { Unknown~key~for~line-numbers }
718
719 }

```

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

```

720 \keys_define:nn { PitonOptions }
721 {
722     indentations-for-Foxit .choices:nn = { true , false }
723     {
724         \tl_if_eq:VnTF \l_keys_value_tl { true }
725         { \@@_define_leading_space_Foxit: }
726         { \@@_define_leading_space_normal: }
727     } ,
728     box .choices:nn = { c , t , b , m }
729     { \str_set_eq:NN \l_@@_box_str \l_keys_choice_tl } ,
730     box .default:n = c ,
731     break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
732     break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,

```

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

```

733     detected-commands .code:n = \@@_detected_commands:n { #1 } ,
734     detected-commands .value_required:n = true ,
735     detected-commands .usage:n = preamble ,

```

```

736 vertical-detected-commands .code:n = \@@_vertical_detected_commands:n { #1 } ,
737 vertical-detected-commands .value_required:n = true ,
738 vertical-detected-commands .usage:n = preamble ,
739 raw-detected-commands .code:n = \@@_raw_detected_commands:n { #1 } ,
740 raw-detected-commands .value_required:n = true ,
741 raw-detected-commands .usage:n = preamble ,
742 detected-beamer-commands .code:n =
743   \@@_error_if_not_in_beamer:
744   \clist_put_right:Nn \l_@@_beamer_commands_clist { #1 } ,
745 detected-beamer-commands .value_required:n = true ,
746 detected-beamer-commands .usage:n = preamble ,
747 detected-beamer-environments .code:n =
748   \@@_error_if_not_in_beamer:
749   \clist_put_right:Nn \l_@@_beamer_environments_clist { #1 } ,
750 detected-beamer-environments .value_required:n = true ,
751 detected-beamer-environments .usage:n = preamble ,

```

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

```

752 begin-escape .code:n =
753   \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
754 begin-escape .value_required:n = true ,
755 begin-escape .usage:n = preamble ,
756
757 after-begin-escape .tl_set:N = \l_@@_after_begin_escape_tl ,
758 after-begin-escape .value_required:n = true ,
759
760 end-escape .code:n =
761   \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
762 end-escape .value_required:n = true ,
763 end-escape .usage:n = preamble ,
764
765 before-end-escape .tl_set:N = \l_@@_before_end_escape_tl ,
766 before-end-escape .value_required:n = true ,
767
768 begin-escape-math .code:n =
769   \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
770 begin-escape-math .value_required:n = true ,
771 begin-escape-math .usage:n = preamble ,
772
773 end-escape-math .code:n =
774   \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
775 end-escape-math .value_required:n = true ,
776 end-escape-math .usage:n = preamble ,
777
778 comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
779 comment-latex .value_required:n = true ,
780 comment-latex .usage:n = preamble ,
781
782 label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
783 label-as-zlabel .usage:n = preamble ,
784
785 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
786 math-comments .usage:n = preamble ,

```

Now, general keys.

```

787 language .code:n =
788   \str_set:Ne \l_piton_language_str { \str_lowercase:n { #1 } } ,
789 language .value_required:n = true ,
790 path .code:n =
791   \seq_clear:N \l_@@_path_seq
792   \clist_map_inline:nn { #1 }
793   {
794     \str_set:Nn \l_tmpa_str { ##1 }
795     \seq_put_right:No \l_@@_path_seq { \l_tmpa_str }

```

```

796     } ,
797     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.

```

798     path .initial:n = . ,
799     path-write .str_set:N = \l_@@_path_write_str ,
800     path-write .value_required:n = true ,
801     font-command .tl_set:N = \l_@@_font_command_tl ,
802     font-command .initial:n = \ttfamily ,
803     font-command .value_required:n = true ,
804     font-command+ .code:n
805     = { \tl_put_right:Nn \l_@@_font_command_tl { #1 } } ,
806     font-command+ .value_required:n = true ,
807     font-command~+ .meta:n = { font-command+ = #1 } ,
808     font-command~+ .value_required:n = true ,
809     gobble .int_set:N = \l_@@_gobble_int ,
810     gobble .default:n = -1 ,
811     auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
812     auto-gobble .value_forbidden:n = true ,
813     env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
814     env-gobble .value_forbidden:n = true ,
815     tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
816     tabs-auto-gobble .value_forbidden:n = true ,
817
818     splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
819
820     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.

```

821     split-separation .tl_set:N = \l_@@_split_separation_tl ,
822     split-separation .value_required:n = true ,
823     split-separation .initial:n = \vspace { \baselineskip } \vspace { -1.25pt } ,
824
825     split-separation+ .code:n =
826     \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
827     split-separation+ .value_required:n = true ,
828     split-separation~+ .meta:n = { split-separation+ = #1 } ,
829     add-to-split-separation .meta:n = { split-separation+ = #1 } ,
830
831     marker .code:n =
832     \bool_lazy_or:nnTF
833     \l_@@_in_PitonInputFile_bool
834     \l_@@_in_PitonOptions_bool
835     { \keys_set:nn { PitonOptions / marker } { #1 } }
836     { \@@_error:n { Invalid~key } } ,
837     marker .value_required:n = true ,
838
839     line-numbers .code:n =
840     \keys_set:nn { PitonOptions / line-numbers } { #1 } ,

```

The following line is mandatory.

```

841     line-numbers .default:n = true ,

```

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).

```

842     splittable .int_set:N = \l_@@_splittable_int ,
843     splittable .default:n = 1 ,
844     splittable .initial:n = 100 ,
845     background-color .code:n =
846     \clist_set:Nn \l_@@_bg_color_clist { #1 }

```

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

```

847     \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
848     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`).

```

849     prompt-background-color .tl_set:N      = \l_@@_prompt_bg_color_tl ,
850     prompt-background-color .value_required:n = true ,
851     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.

```

852     print .bool_set:N = \l_@@_print_bool ,
853     print .value_required:n = true ,
854     print .initial:n = true ,
855
856     width .code:n =
857       \str_if_eq:nnTF { #1 } { min }
858       {
859         \bool_set_true:N \l_@@_minimize_width_bool
860         \dim_zero:N \l_@@_width_dim
861       }
862       {
863         \bool_set_false:N \l_@@_minimize_width_bool
864         \dim_set:Nn \l_@@_width_dim { #1 }
865       } ,
866     width .value_required:n = true ,
867
868     max-width .code:n =
869       \bool_set_true:N \l_@@_minimize_width_bool
870       \dim_set:Nn \l_@@_width_dim { #1 } ,
871     max-width .value_required:n = true ,
872
873     paperclip .code:n =
874       \bool_set_true:N \l_@@_paperclip_bool
875       \tl_if_novalue:nTF { #1 }
876       { \str_set:Nn \l_@@_paperclip_str { } }
877       { \str_set:Nn \l_@@_paperclip_str { #1 } } ,
878
879     annotation .bool_set:N = \l_@@_annotation_bool ,
880
881     write .str_set:N = \l_@@_write_str ,
882     write .value_required:n = true ,
883     no-write .code:n = \str_set_eq:NN \l_@@_write_str \c_empty_str ,
884     no-write .value_forbidden:n = true ,
885     join .code:n =
886       \str_set:Nn \l_@@_join_str { #1 }
887       \seq_if_in:NnF \g_@@_join_seq { #1 }
888       { \seq_gput_right:No \g_@@_join_seq { #1 } } ,
889     join .value_required:n = true ,
890     join-separation .str_set:N = \l_@@_join_separation_str ,
891     join-separation .value_required:n = true ,
892     no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,
893     no-join .value_forbidden:n = true ,
894
895     left-margin .code:n =
896       \str_if_eq:nnTF { #1 } { auto }
897       {
898         \dim_zero:N \l_@@_left_margin_dim
899         \bool_set_true:N \l_@@_left_margin_auto_bool
900       }
901       {

```

```

902         \dim_set:Nn \l_@@_left_margin_dim { #1 }
903         \bool_set_false:N \l_@@_left_margin_auto_bool
904     } ,
905     left-margin .value_required:n = true ,
906
907     right-margin .code:n =
908         \str_if_eq:nnTF { #1 } { auto }
909         {
910             \dim_zero:N \l_@@_right_margin_dim
911             \bool_set_true:N \l_@@_right_margin_auto_bool
912         }
913         {
914             \dim_set:Nn \l_@@_right_margin_dim { #1 }
915             \bool_set_false:N \l_@@_right_margin_auto_bool
916         } ,
917     right-margin .value_required:n = true ,
918
919     tab-size .int_set:N = \l_@@_tab_size_int ,
920     tab-size .value_required:n = true ,
921     tab-size .initial:n = 4 ,
922
923     show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
924     show-spaces .value_forbidden:n = true ,
925
926     show-spaces-in-strings .code:n =
927         \tl_set:Nn \l_@@_space_in_string_tl { \_ } , % U+2423
928     show-spaces-in-strings .value_forbidden:n = true ,
929
930     break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
931     break-lines-in-Piton .initial:n = true ,
932
933     break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
934
935     break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
936     break-lines .value_forbidden:n = true ,
937
938     indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,
939
940     end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
941     end-of-broken-line .value_required:n = true ,
942     end-of-broken-line .initial:n = \hspace* { 0.5em } \textbackslash ,
943
944     continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
945     continuation-symbol .value_required:n = true ,
946     continuation-symbol .initial:n = + ,
947
948     continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
949     continuation-symbol-on-indentation .value_required:n = true ,
950     continuation-symbol-on-indentation .initial:n = $\hookrightarrow$ ; ,
951
952     first-line .code:n = \@@_in_PitonInputFile:n
953         { \int_set:Nn \l_@@_first_line_int { #1 } } ,
954     first-line .value_required:n = true ,
955
956     last-line .code:n = \@@_in_PitonInputFile:n
957         { \int_set:Nn \l_@@_last_line_int { #1 } } ,
958     last-line .value_required:n = true ,
959
960     begin-range .code:n = \@@_in_PitonInputFile:n
961         { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
962     begin-range .value_required:n = true ,
963
964     end-range .code:n = \@@_in_PitonInputFile:n

```

```

965     { \str_set:Nn \l_@@_end_range_str { #1 } } ,
966     end-range .value_required:n = true ,
967
968     range .code:n = \@@_in_PitonInputFile:n
969     {
970         \str_set:Nn \l_@@_begin_range_str { #1 }
971         \str_set:Nn \l_@@_end_range_str { #1 }
972     } ,
973     range .value_required:n = true ,
974
975     env-used-by-split .code:n =
976         \lua_now:n { piton.env_used_by_split = '#1' } ,
977     env-used-by-split .initial:n = Piton ,
978
979     resume .meta:n = line-numbers/resume ,
980
981     unknown .code:n =
982         \@@_unknown_key:nn
983         { PitonOptions }
984         { Unknown~key~for~PitonOptions } ,
985
986     % deprecated
987     all-line-numbers .code:n =
988         \bool_set_true:N \l_@@_line_numbers_bool
989         \bool_set_false:N \l_@@_skip_empty_lines_bool ,
990     rounded-corners .code:n =
991         \AtBeginDocument
992         {
993             \IfPackageLoadedTF { tikz }
994             { \dim_set:Nn \l_@@_rounded_corners_dim { #1 } }
995             { \@@_err_rounded_corners_without_Tikz: }
996         } ,
997     rounded-corners .default:n = 4 pt
998 }
999 \hook_gput_code:nnn { begindocument } { . }
1000 {
1001     \IfPackageLoadedTF { tcolorbox }
1002     {
1003         \pgfkeysifdefined { / tcb / libload / breakable }
1004         {
1005             \keys_define:nn { PitonOptions }
1006             {
1007                 tcolorbox .bool_set:N = \l_@@_tcolorbox_bool ,
1008             }
1009         }
1010         {
1011             \keys_define:nn { PitonOptions }
1012             { tcolorbox .code:n = \@@_error:n { library-breakable-not-loaded } }
1013         }
1014     }
1015     {
1016         \keys_define:nn { PitonOptions }
1017         { tcolorbox .code:n = \@@_error:n { tcolorbox-not-loaded } }
1018     }
1019 }
1020 \cs_new_protected:Npn \@@_err_rounded_corners_without_Tikz:
1021 {
1022     \@@_error:n { rounded-corners-without~Tikz }
1023     \cs_gset:Npn \@@_err_rounded_corners_without_Tikz: { }
1024 }

```



```

1025 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
1026 {
1027   \bool_if:NTF \l_@@_in_PitonInputFile_bool
1028   { #1 }
1029   { \@@_error:n { Invalid~key } }
1030 }

1031 \NewDocumentCommand \PitonOptions { m }
1032 {
1033   \bool_set_true:N \l_@@_in_PitonOptions_bool
1034   \keys_set:nn { PitonOptions } { #1 }
1035   \bool_set_false:N \l_@@_in_PitonOptions_bool
1036 }

```

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.

```

1037 \NewDocumentCommand \@@_fake_PitonOptions { }
1038 { \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.

```

1039 \int_new:N \g_@@_visual_line_int
1040 \cs_new_protected:Npn \@@_incr_visual_line:
1041 { \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`.

```

1042 \cs_new_protected:Npn \@@_print_number_left:
1043 {
1044   \hbox_overlap_left:n
1045   {
1046     \@@_actually_print_number:n { \int_to_arabic:n { \g_@@_visual_line_int } }
1047     \skip_horizontal:N \l_@@_numbers_sep_dim
1048   }
1049 }

```

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`.

```

1050 \cs_new_protected:Npn \@@_print_number_right:
1051 {
1052   \hbox_overlap_left:n
1053   {
1054     \@@_actually_print_number:n { \l_tmpa_tl }
1055     \int_compare:nNnT \l_@@_bg_colors_int > 0
1056     { \skip_horizontal:n { 0.1 em } }
1057   }
1058 }

```

`\@@_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:`

```

1059 \cs_new_protected:Npn \@@_actually_print_number:n #1
1060 {
1061   \group_begin:
1062   \bool_if:NTF \l_@@_lmodern_drawn_bool

```

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

```
1063 { \l_@@_line_numbers_format_tl { \@@_draw_number:e { #1 } } }
1064 {
```

`\space` is mandatory in the “PostScript string” (`\space`) here.

```
1065 \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
1066 \l_@@_line_numbers_format_tl { #1 }
1067 \pdfextension literal { EMC }
1068 }
1069 \group_end:
1070 }
```

2.7 The main commands and environments for the end user

```
1071 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
1072 {
1073   \tl_if_novalue:nTF { #3 }
```

The last argument is provided by curryfication.

```
1074 { \@@_NewPitonLanguage:nnn { #1 } { #2 } }
```

The two last arguments are provided by curryfication.

```
1075 { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
1076 }
```

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.

```
1077 \prop_new:N \g_@@_languages_prop
```

```
1078 \keys_define:nn { NewPitonLanguage }
1079 {
1080   morekeywords .code:n = ,
1081   otherkeywords .code:n = ,
1082   sensitive .code:n = ,
1083   keywordsprefix .code:n = ,
1084   moretexcs .code:n = ,
1085   morestring .code:n = ,
1086   morecomment .code:n = ,
1087   moredelim .code:n = ,
1088   moredirectives .code:n = ,
1089   tag .code:n = ,
1090   alsodigit .code:n = ,
1091   alsoletter .code:n = ,
1092   alsoother .code:n = ,
1093   unknown .code:n = \@@_error:n { Unknown~key-NewPitonLanguage }
1094 }
```

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

```
1095 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
1096 {
```

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}{...}`.

```
1097   \tl_set:N \l_tmpa_tl
1098   {
1099     \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
1100     \str_lowercase:n { #2 }
```

```
1101 }
```

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

```
1102 \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.

```
1103 \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).

```
1104 \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
```

```
1105 }
```

```
1106 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
```

```
1107 {
```

```
1108   \hook_gput_code:nnn { begindocument } { . }
```

```
1109   { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }
```

```
1110 }
```

```
1111 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }
```

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

```
1112 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnnn #1 #2 #3 #4 #5
```

```
1113 {
```

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}{...}`

```
1114   \tl_set:Ne \l_tmpa_tl
```

```
1115   {
```

```
1116     \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
```

```
1117     \str_lowercase:n { #4 }
```

```
1118   }
```

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`.

```
1119   \prop_get:NoNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl
```

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

```
1120   { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
```

```
1121   { \@@_error:n { Language~not~defined } }
```

```
1122 }
```

```
1123 \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.

```
1124   { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
```

```
1125 \cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }
```

```
1126 \NewDocumentCommand { \piton } { }
```

```
1127 { \peek_meaning:NTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }
```

```
1128 \NewDocumentCommand { \@@_piton_standard } { m }
```

```
1129 {
```

```
1130   \group_begin:
```

```
1131   \tl_if_eq:NnF \l_@@_space_in_string_tl { _ }
```

```
1132   {
```

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.

```
1133   \bool_lazy_or:nnT
```

```
1134   \l_@@_break_lines_in_piton_bool
```

```
1135   \l_@@_break_strings_anywhere_bool
```

```
1136   { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
```

```
1137 }
```

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

```
1138 \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:

```
1139 \cs_set_eq:NN \ \ \c_backslash_str
1140 \cs_set_eq:NN \% \c_percent_str
1141 \cs_set_eq:NN \{ \c_left_brace_str
1142 \cs_set_eq:NN \} \c_right_brace_str
1143 \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.

```
1144 \cs_set_eq:cN { ~ } \space
1145 \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
```

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

```
1146 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1147 \tl_set:Ne \l_tmpa_tl
1148 {
1149   \lua_now:e
1150   { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1151   { #1 }
1152 }
1153 \bool_if:NTF \l_@@_show_spaces_bool
1154 { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { _ } } % U+2423
1155 {
1156   \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.

```
1157   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl \space }
1158 }
```

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

```
1159 \if_mode_math:
1160   \text { \l_@@_font_command_tl \l_tmpa_tl }
1161 \else:
1162   \l_@@_font_command_tl \l_tmpa_tl
1163 \fi:
1164 \group_end:
1165 }
```

```
1166 \NewDocumentCommand { \@@_piton_verbatim } { v }
1167 {
1168   \group_begin:
1169   \automatichyphenmode = 1
1170   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
```

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

```
1171 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1172 \tl_set:Ne \l_tmpa_tl
1173 {
1174   \lua_now:e
1175   { piton.Parse('\l_piton_language_str',token.scan_string()) }
1176   { #1 }
1177 }
1178 \bool_if:NT \l_@@_show_spaces_bool
1179 { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { _ } } % U+2423
1180 \if_mode_math:
1181   \text { \l_@@_font_command_tl \l_tmpa_tl }
1182 \else:
1183   \l_@@_font_command_tl \l_tmpa_tl
1184 \fi:
```

```

1185 \group_end:
1186 }

```

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).

```

1187 \cs_new_protected:Npn \@@_piton:n #1
1188 { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }
1189
1190 \cs_new_protected:Npn \@@_piton_i:n #1
1191 {
1192   \group_begin:
1193   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
1194   \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1195   \cs_set:cpn { pitonStyle _ Prompt } { }
1196   \cs_set_eq:NN \@@_leading_space: \space
1197   \cs_set_eq:NN \@@_trailing_space: \space
1198   \tl_set:Nx \l_tmpa_tl
1199   {
1200     \lua_now:e
1201     { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1202     { #1 }
1203   }
1204   \bool_if:NT \l_@@_show_spaces_bool
1205   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { _ } } % U+2423
1206   \@@_replace_spaces:o \l_tmpa_tl
1207   \group_end:
1208 }

```

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

```

1209 \cs_new_protected:Npn \@@_pre_composition:
1210 {
1211   \dim_compare:nNt \l_@@_width_dim = \c_zero_dim
1212   {
1213     \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).

```

1214   \str_if_empty:NF \l_@@_box_str
1215   { \bool_set_true:N \l_@@_minimize_width_bool }
1216 }

```

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).

```

1217   \dim_set:Nn \l_@@_listing_width_dim
1218   {
1219     \bool_if:NtF \l_@@_tcolorbox_bool
1220     {
1221       \l_@@_width_dim -
1222       ( \kvtcb@left@rule
1223       + \kvtcb@leftupper
1224       + \kvtcb@boxsep * 2
1225       + \kvtcb@rightupper
1226       + \kvtcb@right@rule )
1227     }
1228     { \l_@@_width_dim }
1229   }
1230   \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1231   \automatichyphenmode = 1

```

```

1232 \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1233 \g_@@_def_vertical_commands_tl
1234 \int_gzero:N \g_@@_line_int
1235 \int_gzero:N \g_@@_nb_lines_int
1236 \dim_zero:N \parindent
1237 \dim_zero:N \lineskip
1238 \dim_zero:N \parskip
1239
1240 \seq_gclear:N \g_@@_visual_line_numbers_seq
1241
1242 \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

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

```

1243 \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1244 { \bool_set_true:N \l_@@_bg_bool }
1245 \bool_gset_false:N \g_@@_rowcolor_inside_bool
1246 \IfPackageLoadedTF { zref-base }
1247 {
1248   \bool_if:NTF \g_@@_label_as_zlabel_bool
1249   { \cs_set_eq:NN \label \@@_zlabel:n }
1250   { \cs_set_eq:NN \label \@@_label:n }
1251   \cs_set_eq:NN \zlabel \@@_zlabel:n
1252 }
1253 { \cs_set_eq:NN \label \@@_label:n }
1254 \l_@@_font_command_tl
1255 }

```

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`.

```

1256 \cs_new_protected:Npn \@@_compute_margin:N #1
1257 {
1258   \use:e
1259   {
1260     \bool_if:NTF \l_@@_skip_empty_lines_bool
1261     { \lua_now:n { piton.CountNonEmptyLines(token.scan_argument()) } }
1262     { \lua_now:n { piton.CountLines(token.scan_argument()) } }
1263     { \l_@@_listing_tl }
1264   }
1265   \hbox_set:Nn \l_tmpa_box
1266   {
1267     \l_@@_line_numbers_format_tl
1268     \int_to_arabic:n
1269     {
1270       \g_@@_visual_line_int
1271       +
1272       \bool_if:NTF \l_@@_skip_empty_lines_bool
1273       { \l_@@_nb_non_empty_lines_int }
1274       { \g_@@_nb_lines_int }
1275     }
1276   }
1277   \dim_set:Nn #1 { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1278 }

```

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.

```

1279 \cs_new_protected:Npn \@@_compute_code_width:
1280 {
1281   \dim_set:Nn \l_@@_code_width_dim
1282   {
1283     \l_@@_listing_width_dim
1284     -
1285     (
1286       \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1287       {
1288         \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1289         { \l_@@_left_margin_dim }
1290         { 0.5 em }
1291       }
1292       +
1293       \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1294       { \l_@@_right_margin_dim }
1295       { 0.5 em }
1296     )
1297     { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1298   }
1299 }

```

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:`.

```

1300 \cs_new_protected:Npn \@@_recompute_listing_width:
1301 {
1302   \dim_set:Nn \l_@@_listing_width_dim
1303   {
1304     \box_wd:N \g_@@_output_box
1305     +
1306     \int_compare:nNnTF \l_@@_bg_colors_int > \c_zero_int
1307     {
1308       \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1309       { \l_@@_left_margin_dim }
1310       { 0.5 em }
1311     }
1312     +
1313     \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1314     { \l_@@_right_margin_dim }
1315     { 0.5 em }
1316   }
1317   { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1318 }

```

```

1319 \cs_new_protected:Npn \@@_store_body:n #1
1320 {

```

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

```

1321   \tl_set:Nc \obeyedline { \char_generate:nn { 13 } { 11 } }
1322   \tl_set:Nc \l_@@_listing_tl { #1 }
1323   \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1324 }

```

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

```

1325 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nnnnn
1326 {

```

```

1327 \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1328 {
1329   \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1330   #4
1331   \@@_pre_composition:
1332   \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1333   {
1334     \int_gset:Nn \g_@@_visual_line_int
1335     { \l_@@_number_lines_start_int - 1 }
1336   }
1337   \bool_if:NT \g_@@_beamer_bool
1338   { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1339   \bool_if:NT \g_@@_footnote_bool \savenotes
1340   \@@_composition:
1341   \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1342   { \@@_create_paperclip_or_annotation: }
1343   \bool_if:NT \g_@@_footnote_bool \endsavenotes
1344   #5
1345 }
1346 { \ignorespacesafterend }
1347 }

```

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

```

1348 \cs_new_protected:Npn \@@_create_paperclip_or_annotation:
1349 {
1350   \marginalia [ pos = right ]
1351   {
1352     \vspace* { - 0.8 em }
1353     \hbox:n
1354     {
1355       \vrule-height~0pt~depth~12pt~width~0pt
1356       \bool_if:NT \l_@@_annotation_bool
1357       {
1358         \lua_now:n
1359         {

```

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!

```

1360         pdf.immediateobj
1361         ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1362       }
1363     \pdfextension annot~width~5pt~height~10pt~depth~0pt
1364     {
1365       /Subtype /Text
1366       /Contents~\pdf_object_ref_last:
1367       /Name /Note
1368       /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).

```

1369       /ReplyType /Group

```

Adds the bit 10 which means `LockedContents`.

```

1370       /F~512
1371       /C [0.8~0.8~0.8]
1372     }
1373     \hspace* { 7 mm }
1374   }
1375   \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1376 }
1377 }
1378 }

```

```

1379 \cs_new_protected:Npn \@@_create_paperclip:

```



```

1380 {
1381   \str_if_empty:NT \l_@@_paperclip_str
1382   {
1383     \int_gincr:N \g_@@_paperclip_int
1384     \str_set:Ne \l_@@_paperclip_str { listing\_int\_use:N \g_@@_paperclip_int .txt }
1385   }

```

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

```

1386   \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1387   \box_move_down:nn
1388   { 10 pt }
1389   {
1390     \hbox:n
1391     {
1392       \pdfextension annot~width~10pt~height~20pt~depth~0pt
1393       {
1394         /Subtype /FileAttachment
1395         /Name /Paperclip
1396         /F~8 % no zoom

```

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

```

1397       /Contents (The~computer~listing)
1398       /FS <<
1399         /Type /Filespec
1400         /F (\l_@@_paperclip_str)
1401         /EF << /F~\pdf\_object\_ref\_last: >>
1402         /AFRelationship /Supplement
1403       >>
1404     }
1405   }
1406 }
1407 }

```

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

```

1408 \NewDocumentCommand { \NewPitonEnvironment } { }
1409 { \@@_DefinePitonEnvironment:nnnnn { New } }

1410 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1411 { \@@_DefinePitonEnvironment:nnnnn { Declare } }

1412 \NewDocumentCommand { \RenewPitonEnvironment } { }
1413 { \@@_DefinePitonEnvironment:nnnnn { Renew } }

1414 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1415 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

1416 \cs_new_protected:Npn \@@_translate_beamer_env:n
1417 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1418 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1419 \cs_new_protected:Npn \@@_composition:
1420 {
1421   \str_if_empty:NT \l_@@_box_str
1422   {
1423     \mode_if_vertical:F
1424     { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1425   }

1426   \bool_if:NT \l_@@_line_numbers_bool
1427   {
1428     \bool_lazy_and:nnT
1429     { \l_@@_left_margin_auto_bool }
1430     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { left } }
1431     { \@@_compute_margin:N \l_@@_left_margin_dim }
1432     \bool_lazy_and:nnT
1433     { \l_@@_right_margin_auto_bool }

```

```

1434         { \str_if_eq_p:ee \l_@@_line_numbers_position_str { right } }
1435         { \@@_compute_margin:N \l_@@_right_margin_dim }
1436     }
1437 \lua_now:e
1438 {
1439     piton.join_separation = "\l_@@_join_separation_str"
1440     piton.join = "\l_@@_join_str"
1441     piton.write = "\l_@@_write_str"
1442     piton.path_write = "\l_@@_path_write_str"
1443 }
1444 \noindent
1445 \bool_if:NTF \l_@@_print_bool
1446 {

```

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.

```

1447     \bool_if:NTF \l_@@_split_on_empty_lines_bool
1448     { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1449     {
1450         \@@_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).

```

1451         \bool_if:NTF \l_@@_tcolorbox_bool
1452         {
1453             \str_if_empty:NTF \l_@@_box_str
1454             \@@_composition_iii:
1455             \@@_composition_iv:
1456         }
1457         {
1458             \str_if_empty:NTF \l_@@_box_str
1459             \@@_composition_i:
1460             \@@_composition_ii:
1461         }
1462     }
1463 }
1464 { \@@_gobble_parse_no_print:o \l_@@_listing_tl }
1465 }

```

`\@@_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=...`

```

1466 \cs_new_protected:Npn \@@_composition_i:
1467 {

```

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.

```

1468     \box_clear:N \g_tmpa_box

```

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

```

1469     \box_clear_new:N \g_@@_line_box

```

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

```

1470     \vbox_set:Nn \l_tmpa_box
1471     {
1472         \vbox_unpack_drop:N \g_@@_output_box
1473         \bool_gset_false:N \g_tmpa_bool
1474         \unskip \unskip
1475         \bool_gset_false:N \g_tmpa_bool
1476         \bool_do_until:nn \g_tmpa_bool
1477         {

```

```

1478      \unskip \unskip \unskip
1479      \unpenalty \unkern
1480      \box_set_to_last:N \l_@@_line_box
1481      \box_if_empty:NTF \l_@@_line_box
1482      { \bool_gset_true:N \g_tmpa_bool }
1483      {
1484          \vbox_gset:Nn \g_tmpa_box
1485          {
1486              \vbox_unpack:N \g_tmpa_box
1487              \box_use:N \l_@@_line_box
1488          }
1489      }
1490  }
1491 }

```

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

```

1492      \bool_gset_false:N \g_tmpa_bool
1493      \int_zero:N \g_@@_line_int
1494      \bool_do_until:nn \g_tmpa_bool
1495      {

```

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`.

```

1496          \vbox_gset:Nn \g_tmpa_box
1497          {
1498              \vbox_unpack_drop:N \g_tmpa_box
1499              \box_gset_to_last:N \g_@@_line_box
1500          }

```

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.

```

1501          \box_if_empty:NTF \g_@@_line_box
1502          { \bool_gset_true:N \g_tmpa_bool }
1503          {
1504              \box_use:N \g_@@_line_box
1505              \int_gincr:N \g_@@_line_int
1506              \par
1507              \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`.

```

1508          \@@_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.

```

1509          \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int }
1510          { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int } }
1511          \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1512          { \mode_leave_vertical: }
1513      }
1514  }
1515  \skip_vertical:n { 2.5 pt }
1516 }

```

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

```

1517 \cs_new_protected:Npn \@@_composition_ii:
1518 {
1519     \use:e { \begin { minipage } [ \l_@@_box_str ] }
1520     { \l_@@_listing_width_dim }

```

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

```

1521     \vbox_unpack:N \g_@@_output_box

```

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

```

1522     \kern 2.5 pt
1523     \end { minipage }
1524 }

```

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

```

1525 \cs_new_protected:Npn \@@_composition_iii:
1526 {
1527   \use:e
1528   {
1529     \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.

```

1530       [ breakable , text~width = \l_@@_listing_width_dim ]
1531   }
1532   \par
1533   \vbox_unpack:N \g_@@_output_box
1534   \end { tcolorbox }
1535 }

```

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

```

1536 \cs_new_protected:Npn \@@_composition_iv:
1537 {
1538   \use:e
1539   {
1540     \begin { tcolorbox }
1541     [
1542       hbox ,
1543       text~width = \l_@@_listing_width_dim ,
1544       nobeforeafter ,
1545       box~align =
1546         \str_case:Nn \l_@@_box_str
1547         {
1548           t { top }
1549           b { bottom }
1550           c { center }
1551           m { center }
1552         }
1553     ]
1554   }
1555   \box_use:N \g_@@_output_box
1556   \end { tcolorbox }
1557 }

```

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.

```

1558 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1559 {
1560   \int_case:nn
1561   {
1562     \lua_now:e
1563     {
1564       tex.sprint
1565       ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1566     }
1567   }
1568   { 1 { \penalty 100 } 2 \nobreak }
1569 }

```

`\@@_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`.

```

1570 \cs_new_protected:Npn \@@_create_output_box:
1571 {

```

```

1572 \@@_compute_code_width:
1573 \vbox_gset:Nn \g_@@_output_box
1574 { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1575 \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1576 \bool_lazy_any:nT
1577 {
1578   { \int_compare_p:nNn \l_@@_bg_colors_int > \c_zero_int }
1579   { \g_@@_rowcolor_inside_bool }
1580   {
1581     \l_@@_line_numbers_bool
1582     &&
1583     \str_if_eq_p:ee \l_@@_line_numbers_position_str { right }
1584   }
1585 }
1586 \@@_add_bg_and_right_nb_to_output_box:
1587 }

```

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:.`

```

1588 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_output_box:
1589 {
1590   \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`.

```

1591   \vbox_set:Nn \l_tmpa_box
1592   {
1593     \vbox_unpack_drop:N \g_@@_output_box

```

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

```

1594     \bool_gset_false:N \g_tmpa_bool
1595     \unskip \unskip

```

We begin the loop.

```

1596     \bool_do_until:nn \g_tmpa_bool
1597     {
1598       \unskip \unskip \unskip
1599       \int_set_eq:NN \l_tmpa_int \lastpenalty
1600       \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 the L3 Programming Layer). Of course, it would be interesting to replace that programming by a programming in Lua of LuaTeX...

```

1601       \box_set_to_last:N \l_@@_line_box
1602       \box_if_empty:NTF \l_@@_line_box
1603       { \bool_gset_true:N \g_tmpa_bool }
1604       {

```

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

```

1605       \vbox_gset:Nn \g_@@_output_box
1606       {

```

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`.

```

1607         \@@_add_bg_and_right_nb_to_line_and_use:
1608         \kern -2.5 pt
1609         \penalty \l_tmpa_int
1610         \vbox_unpack:N \g_@@_output_box
1611       }
1612     }
1613     \int_gdecr:N \g_@@_line_int
1614   }
1615 }

```

```
1616 }
```

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

```
1617 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1618 {
1619   \lua_now:e
1620   {
1621     piton.GobbleParseNoPrint
1622     (
1623       '\l_piton_language_str' ,
1624       \int_use:N \l_@@_gobble_int ,
1625       token.scan_argument ( )
1626     )
1627   }
1628 }
1629 \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.

```
1630 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1631 {
1632   \lua_now:e
1633   {
1634     piton.RetrieveGobbleParse
1635     (
1636       '\l_piton_language_str' ,
1637       \int_use:N \l_@@_gobble_int ,
1638       \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1639       { \int_eval:n { - \l_@@_splittable_int } }
1640       { \int_use:N \l_@@_splittable_int } ,
1641       token.scan_argument ( )
1642     )
1643   }
1644 }
1645 \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.

```
1646 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1647 {
1648   \lua_now:e
1649   {
1650     piton.RetrieveGobbleSplitParse
1651     (
1652       '\l_piton_language_str' ,
1653       \int_use:N \l_@@_gobble_int ,
1654       \int_use:N \l_@@_splittable_int ,
1655       token.scan_argument ( )
1656     )
1657   }
1658 }
1659 \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`.

```
1660 \bool_if:NTF \g_@@_beamer_bool
1661 {
1662   \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1663   {
1664     \keys_set:nn { PitonOptions } { #2 }
```

```

1665     \begin { actionenv } < #1 >
1666   }
1667   { \end { actionenv } }
1668 }
1669 {
1670   \NewPitonEnvironment { Piton } { 0 { } }
1671   { \keys_set:nn { PitonOptions } { #1 } }
1672   { }
1673 }

1674 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1675 {
1676   \mode_if_vertical:F { \par }
1677   \group_begin:
1678   \seq_concat:NNN
1679     \l_file_search_path_seq
1680     \l_@@_path_seq
1681     \l_file_search_path_seq
1682   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1683   {
1684     \@@_input_file:nn { #1 } { #2 }
1685     #4
1686   }
1687   { #5 }
1688   \group_end:
1689 }

1690 \cs_new_protected:Npn \@@_unknown_file:n #1
1691 { \msg_error:nnn { piton } { Unknown-file } { #1 } }

1692 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1693 {
1694   \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1695 }

```

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

```

1696   \iow_log:n { No-file-#3 }
1697   \@@_unknown_file:n { #3 }
1698 }
1699 }
1700 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1701 {
1702   \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1703 }

```

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

```

1704   \iow_log:n { No-file-#3 }
1705   \@@_unknown_file:n { #3 }
1706 }
1707 }
1708 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1709 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```

1710 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1711 {

```

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 >).

```

1712   \tl_if_novalue:nF { #1 }
1713   {
1714     \bool_if:NTF \g_@@_beamer_bool
1715       { \begin { uncoverenv } < #1 > }
1716       { \@@_error_or_warning:n { overlay~without~beamer } }
1717   }
1718   \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.

```

1719 \iow_log:e { (\l_@@_file_name_str) }
1720 \int_zero_new:N \l_@@_first_line_int
1721 \int_zero_new:N \l_@@_last_line_int
1722 \int_set_eq:NN \l_@@_last_line_int \c_max_int
1723 \bool_set_true:N \l_@@_in_PitonInputFile_bool
1724 \keys_set:nn { PitonOptions } { #2 }
1725 \bool_if:NT \l_@@_line_numbers_absolute_bool
1726 { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1727 \bool_if:nTF
1728 {
1729   (
1730     \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1731     || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1732   )
1733   && ! \str_if_empty_p:N \l_@@_begin_range_str
1734 }
1735 {
1736   \@@_error_or_warning:n { bad-range-specification }
1737   \int_zero:N \l_@@_first_line_int
1738   \int_set_eq:NN \l_@@_last_line_int \c_max_int
1739 }
1740 {
1741   \str_if_empty:NF \l_@@_begin_range_str
1742   {
1743     \@@_compute_range:
1744     \bool_lazy_or:nnT
1745       \l_@@_marker_include_lines_bool
1746       { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1747     {
1748       \int_decr:N \l_@@_first_line_int
1749       \int_incr:N \l_@@_last_line_int
1750     }
1751   }
1752 }
1753 \@@_pre_composition:
1754 \bool_if:NT \l_@@_line_numbers_absolute_bool
1755 { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1756 \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1757 {
1758   \int_gset:Nn \g_@@_visual_line_int
1759   { \l_@@_number_lines_start_int - 1 }
1760 }

```

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

```

1761 \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1762 { \int_gzero:N \g_@@_visual_line_int }
1763 \lua_now:e
1764 {

```

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

```

1765 piton.ReadFile(
1766   '\l_@@_file_name_str' ,
1767   \int_use:N \l_@@_first_line_int ,
1768   \int_use:N \l_@@_last_line_int )
1769 }
1770 \@@_composition:
1771 \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.

```

1772 \tl_if_novalue:nF { #1 }

```



```

1773     { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1774 }

```

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

```

1775 \cs_new_protected:Npn \@@_compute_range:
1776 {

```

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

```

1777   \str_set:N \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1778   \str_set:N \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`.

```

1779   \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1780   \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str

1781   \lua_now:e
1782   {
1783       piton.ComputeRange
1784       ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1785   }
1786 }

```

2.8 The styles

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

```

1787 \NewDocumentCommand { \PitonStyle } { m }
1788 {
1789     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1790     { \use:c { pitonStyle _ #1 } }
1791 }

```

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”.

```

1792 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1793 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1794 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1795 {
1796     \str_clear_new:N \l_@@_SetPitonStyle_option_str
1797     \str_set:N \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1798     \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1799     { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1800     \keys_set:nn { piton / Styles } { #2 }
1801 }

1802 \cs_new_protected:Npn \@@_math_scantokens:n #1
1803 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

```

```

1804 \clist_new:N \g_@@_styles_clist
1805 \clist_gset:Nn \g_@@_styles_clist
1806 {
1807     Comment ,
1808     Comment.Internal ,
1809     Comment.LaTeX ,
1810     Discard ,
1811     Delim ,
1812     Exception ,
1813     FormattingType ,
1814     Identifier.Internal ,
1815     Identifier ,

```

```

1816 InitialValues ,
1817 Interpol.Inside ,
1818 Keyword ,
1819 Keyword.Governing ,
1820 Keyword.Constant ,
1821 Keyword2 ,
1822 Keyword3 ,
1823 Keyword4 ,
1824 Keyword5 ,
1825 Keyword6 ,
1826 Keyword7 ,
1827 Keyword8 ,
1828 Keyword9 ,
1829 Name.Builtin ,
1830 Name.Class ,
1831 Name.Constructor ,
1832 Name.Decorator ,
1833 Name.Field ,
1834 Name.Function ,
1835 Name.Module ,
1836 Name.Namespace ,
1837 Name.Table ,
1838 Name.Type ,
1839 Number ,
1840 Number.Internal ,
1841 Operator ,
1842 Operator.Word ,
1843 Preproc ,
1844 Prompt ,
1845 Punct ,
1846 String.Doc ,
1847 String.Doc.Internal ,
1848 String.Interpol ,
1849 String.Long ,
1850 String.Long.Internal ,
1851 String.Short ,
1852 String.Short.Internal ,
1853 Tag ,
1854 TypeParameter ,
1855 UserFunction ,

```

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

```

1856 TypeExpression ,

```

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

```

1857 Directive
1858 }

1859 \clist_map_inline:Nn \g_@@_styles_clist
1860 {
1861   \keys_define:nn { piton / Styles }
1862   {
1863     #1 .value_required:n = true ,
1864     #1 .code:n =
1865       \tl_set:cn
1866       {
1867         pitonStyle _
1868         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1869         { \l_@@_SetPitonStyle_option_str _ }
1870         #1
1871       }
1872     { ##1 }
1873   }
1874 }
1875

```

```

1876 \keys_define:nn { piton / Styles }
1877 {
1878   String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1879   String      .value_required:n = true ,
1880   Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1881   Comment.Math .value_required:n = true ,
1882   unknown     .code:n = \@@_unknown_style:
1883 }

```

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.

```

1884 \cs_new_protected:Npn \@@_unknown_style:
1885 {
1886   \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1887   {
1888     \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1889     \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`.

```

1890     \bool_lazy_and:nnTF
1891     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1892     {
1893       \str_if_eq_p:Vn \l_tmpa_str { Module }
1894       ||
1895       \str_if_eq_p:Vn \l_tmpa_str { Type }
1896     }

```

Now, we will create a new style.

```

1897     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1898     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1899   }
1900   { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1901 }

```

```

1902 \SetPitonStyle[OCaml]
1903 {
1904   TypeExpression =
1905   {
1906     \SetPitonStyle [ OCaml ]
1907     {
1908       Identifier = \PitonStyle { Name.Type } ,
1909       Name.Builtin = \PitonStyle { Name.Type}
1910     }
1911     \@@_piton:n
1912   }
1913 }

```

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`.

```

1914 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that `clist`.

```

1915 \clist_gsort:Nn \g_@@_styles_clist
1916 {
1917   \str_compare:nNnTF { #1 } < { #2 }
1918     \sort_return_same:
1919     \sort_return_swapped:
1920 }

```

```

1921 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1922
1923 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1924
1925 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1926 {
1927   \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`.

```

1928   \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl \space
1929   \seq_clear:N \l_tmpa_seq
1930   \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1931   \seq_use:Nn \l_tmpa_seq { \- }
1932 }

```

```

1933 \cs_new_protected:Npn \@@_comment:n #1
1934 { \PitonStyle { Comment } { \PitonSpaceSubstitute { #1 } } }

```

We use a standard name for the following command (and not an internal name of L3 such as `\@@_space_substitute:n` because it will be inserted in the main LaTeX stream by Lua when `morecomment` is used in `\NewPitonLanguage`.

We should probably change that with an “internal style”.

```

1935 \cs_new_protected:Npn \PitonSpaceSubstitute #1 % noqa
1936 {
1937   \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1938   {
1939     \tl_set:Nn \l_tmpa_tl { #1 }
1940     \tl_replace_all:Nvn \l_tmpa_tl
1941       \c_catcode_other_space_tl
1942       \@@_breakable_space:
1943     \l_tmpa_tl
1944   }
1945   { #1 }
1946 }

```

```

1947 \cs_new_protected:Npn \@@_string_long:n #1
1948 {
1949   \PitonStyle { String.Long }
1950   {
1951     \bool_if:NTF \l_@@_break_strings_anywhere_bool
1952     { \@@_actually_break_anywhere:n { #1 } }
1953     {

```

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.

```

1954       \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1955       {
1956         \tl_set:Nn \l_tmpa_tl { #1 }
1957         \tl_replace_all:Nvn \l_tmpa_tl
1958           \c_catcode_other_space_tl
1959           \@@_breakable_space:
1960         \l_tmpa_tl
1961       }
1962       { #1 }
1963     }
1964   }
1965 }

```

```

1966 \cs_new_protected:Npn \@@_string_short:n #1
1967 {
1968   \PitonStyle { String.Short }
1969   {
1970     \bool_if:NT \l_@@_break_strings_anywhere_bool
1971     { \@@_actually_break_anywhere:n }
1972     { #1 }
1973   }
1974 }
1975 \cs_new_protected:Npn \@@_string_doc:n #1
1976 {
1977   \PitonStyle { String.Doc }
1978   {
1979     \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1980     {
1981       \tl_set:Nn \l_tmpa_tl { #1 }
1982       \tl_replace_all:NVn \l_tmpa_tl
1983       \c_catcode_other_space_tl
1984       \@@_breakable_space:
1985       \l_tmpa_tl
1986     }
1987     { #1 }
1988   }
1989 }
1990 \cs_new_protected:Npn \@@_number:n #1
1991 {
1992   \PitonStyle { Number }
1993   {
1994     \bool_if:NT \l_@@_break_numbers_anywhere_bool
1995     { \@@_actually_break_anywhere:n }
1996     { #1 }
1997   }
1998 }

```

2.9 The initial styles

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

```

1999 \SetPitonStyle
2000 {
2001   Comment           = \color [ HTML ] { 0099FF } \itshape ,
2002   Comment.Internal  = \@@_comment:n ,
2003   Exception         = \color [ HTML ] { CC0000 } ,
2004   Keyword           = \color [ HTML ] { 006699 } \bfseries ,
2005   Keyword.Governing = \color [ HTML ] { 006699 } \bfseries ,
2006   Keyword.Constant  = \color [ HTML ] { 006699 } \bfseries ,
2007   Name.Builtin      = \color [ HTML ] { 336666 } ,
2008   Name.Decorator     = \color [ HTML ] { 9999FF } ,
2009   Name.Class        = \color [ HTML ] { 00AA88 } \bfseries ,
2010   Name.Function     = \color [ HTML ] { CC00FF } ,
2011   Name.Namespace    = \color [ HTML ] { 00CCFF } ,
2012   Name.Constructor  = \color [ HTML ] { 006000 } \bfseries ,
2013   Name.Field        = \color [ HTML ] { AA6600 } ,
2014   Name.Module       = \color [ HTML ] { 0060A0 } \bfseries ,
2015   Name.Table        = \color [ HTML ] { 309030 } ,
2016   Number            = \color [ HTML ] { FF6600 } ,
2017   Number.Internal   = \@@_number:n ,
2018   Operator          = \color [ HTML ] { 555555 } ,
2019   Operator.Word     = \bfseries ,
2020   String            = \color [ HTML ] { CC3300 } ,
2021   String.Long.Internal = \@@_string_long:n ,
2022   String.Short.Internal = \@@_string_short:n ,

```

```

2023 String.Doc.Internal = \@@_string_doc:n ,
2024 String.Doc          = \color [ HTML ] { CC3300 } \itshape ,
2025 String.Interpol     = \color [ HTML ] { AA0000 } ,
2026 Comment.LaTeX       = \normalfont \color [ rgb ] { .468, .532, .6 } ,
2027 Name.Type           = \color [ HTML ] { 336666 } ,
2028 InitialValues       = \@@_piton:n ,
2029 Interpol.Inside      = { \l_@@_font_command_tl \@@_piton:n } ,
2030 TypeParameter       = \color [ HTML ] { 336666 } \itshape ,
2031 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).

```

2032 Identifier.Internal = \@@_identifier:n ,
2033 Identifier          = ,
2034 Directive          = \color [ HTML ] { AA6600 } ,
2035 Tag                = \colorbox { gray!10 } ,
2036 UserFunction       = \PitonStyle { Identifier } ,
2037 Prompt            = ,
2038 Discard            = \use_none:n
2039 }

```

2.10 Styles specific to the language expl

```

2040 \clist_new:N \g_@@_expl_styles_clist
2041 \clist_gset:Nn \g_@@_expl_styles_clist
2042 {
2043   Scope.l ,
2044   Scope.g ,
2045   Scope.c
2046 }
2047 \clist_map_inline:Nn \g_@@_expl_styles_clist
2048 {
2049   \keys_define:nm { piton / Styles }
2050   {
2051     #1 .value_required:n = true ,
2052     #1 .code:n =
2053       \tl_set:cn
2054       {
2055         pitonStyle _
2056         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
2057         { \l_@@_SetPitonStyle_option_str _ }
2058         #1
2059       }
2060     { ##1 }
2061   }
2062 }
2063 \SetPitonStyle [ expl ]
2064 {
2065   Scope.l      = ,
2066   Scope.g      = \bfseries ,
2067   Scope.c      = \slshape ,
2068   Type.bool    = \color [ HTML ] { AA6600 } ,
2069   Type.box     = \color [ HTML ] { 267910 } ,
2070   Type.clist   = \color [ HTML ] { 309030 } ,
2071   Type.fp      = \color [ HTML ] { FF3300 } ,
2072   Type.int     = \color [ HTML ] { FF6600 } ,
2073   Type.seq     = \color [ HTML ] { 309030 } ,
2074   Type.skip    = \color [ HTML ] { 0CC060 } ,
2075   Type.str     = \color [ HTML ] { CC3300 } ,
2076   Type.tl      = \color [ HTML ] { AA2200 } ,

```

```

2077 Module.bool      = \color [ HTML ] { AA6600 } ,
2078 Module.box       = \color [ HTML ] { 267910 } ,
2079 Module.cs        = \bfseries \color [ HTML ] { 006699 } ,
2080 Module.exp       = \bfseries \color [ HTML ] { 404040 } ,
2081 Module.hbox      = \color [ HTML ] { 267910 } ,
2082 Module.prg       = \bfseries ,
2083 Module.clist     = \color [ HTML ] { 309030 } ,
2084 Module.fp        = \color [ HTML ] { FF3300 } ,
2085 Module.int       = \color [ HTML ] { FF6600 } ,
2086 Module.seq       = \color [ HTML ] { 309030 } ,
2087 Module.skip      = \color [ HTML ] { 0CC060 } ,
2088 Module.str       = \color [ HTML ] { CC3300 } ,
2089 Module.tl        = \color [ HTML ] { AA2200 } ,
2090 Module.vbox      = \color [ HTML ] { 267910 }
2091 }

```

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).

```

2092 \hook_gput_code:nnn { begindocument } { . }
2093 {
2094   \bool_if:NT \g_@@_math_comments_bool
2095     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
2096 }

```

2.11 Highlighting some identifiers

```

2097 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
2098 {
2099   \clist_set:Nn \l_tmpa_clist { #2 }
2100   \tl_if_novalue:nTF { #1 }
2101     {
2102       \clist_map_inline:Nn \l_tmpa_clist
2103         { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
2104     }
2105     {
2106       \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
2107       \str_if_eq:onT \l_tmpa_str { current-language }
2108         { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
2109       \clist_map_inline:Nn \l_tmpa_clist
2110         { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
2111     }
2112 }
2113 \cs_new_protected:Npn \@@_identifier:n #1
2114 {
2115   \str_set:Nn \l_tmpa_str { #1 }
2116   \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2117     {
2118       \cs_if_exist_use:cF { PitonIdentifier _ \l_tmpa_str }
2119         { \PitonStyle { Identifier } }
2120     }
2121   { #1 }
2122 }

```

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 (we define it directly and we short-cut the function `\SetPitonStyle`).

```

2123 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
2124 {

```

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

```
2125 { \PitonStyle { Name.Function } { #1 } }
2126 \str_set:Nn \l_tmpa_str { #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}`.

```
2127 \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2128 { \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`.**

```
2129 \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
2130 { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
2131 \seq_gput_right:co { g_@@_functions _ \l_piton_language_str _ seq } \l_tmpa_str
```

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

```
2132 \seq_if_in:NoF \g_@@_languages_seq { \l_piton_language_str }
2133 { \seq_gput_left:No \g_@@_languages_seq { \l_piton_language_str } }
2134 }
```

```
2135 \NewDocumentCommand \PitonClearUserFunctions { ! o }
2136 {
2137   \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.

```
2138 { \@@_clear_all_functions: }
2139 { \@@_clear_list_functions:n { #1 } }
2140 }
```

```
2141 \cs_new_protected:Npn \@@_clear_list_functions:n #1
2142 {
2143   \clist_set:Nn \l_tmpa_clist { #1 }
2144   \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
2145   \clist_map_inline:nn { #1 }
2146   { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
2147 }
```

```
2148 \cs_new_protected:Npn \@@_clear_functions_i:n #1
2149 { \@@_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).

```
2150 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2151 {
2152   \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2153   {
2154     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2155     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }
2156     \seq_gclear:c { g_@@_functions _ #1 _ seq }
2157   }
2158 }
2159 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }
```

```
2160 \cs_new_protected:Npn \@@_clear_functions:n #1
2161 {
2162   \@@_clear_functions_i:n { #1 }
2163   \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2164 }
```

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


```

2165 \cs_new_protected:Npn \@@_clear_all_functions:
2166 {
2167   \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2168   \seq_gclear:N \g_@@_languages_seq
2169 }

```

```

2170 \AtEndDocument
2171 {

```

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

```

2172   \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.

```

2173   \IfPDFManagementActiveTF
2174   { \@@_join_files: }
2175   { \@@_join_files_legacy: }
2176 }

```

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

```

2177 \cs_new_protected:Npn \@@_join_files:
2178 {
2179   \seq_map_inline:Nn \g_@@_join_seq
2180   {
2181     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2182     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2183     \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2184     {
2185       <<
2186         /Type /Filespec
2187         /UF <\l_tmpa_str>
2188         /EF << /F~\pdf_object_ref_last: >>
2189         /Desc (Computer~listing)
2190         /AFRelationship /Supplement
2191       >>
2192     }
2193   }
2194 }

```

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

```

2195 \cs_new_protected:Npn \@@_join_files_legacy:
2196 {
2197   \seq_map_inline:Nn \g_@@_join_seq
2198   {
2199     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2200     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2201     \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 0pt height 0pt depth 0pt`.

```

2202     {
2203       /Subtype /FileAttachment
2204       /F~2
2205       /Name /Paperclip
2206       /Contents (Computer~listing)
2207       /FS <<
2208         /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).

```
2209 /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`).

```
2210 /EF << /F~\pdf_object_ref_last: >>
2211 /AFRelationship /Supplement
2212 >>
2213 }
2214 }
2215 }
```

2.12 Spaces of indentation

```
2216 \cs_new_protected:Npn \@@_define_leading_space_normal:
2217 {
2218   \cs_set_protected:Npn \@@_leading_space:
2219   {
2220     \int_gincr:N \g_@@_indentation_int
```

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

```
2221   \hbox:n { ~ }
2222 }
2223 }
2224 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2225 {
2226   \cs_set_protected:Npn \@@_leading_space:
2227   {
2228     \int_gincr:N \g_@@_indentation_int
2229     \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2230     {
2231       \color { white }
2232       \transparent { 0 }
2233       . % previously : □ U+2423
2234     }
2235     \pdfextension literal { EMC }
2236   }
2237 }
2238 \@@_define_leading_space_Foxit:
```

2.13 Security

```
2239 \AddToHook { env / piton / before }
2240 { \@@_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.

```
2241 \cs_new_protected:Npn \@@_unknown_key:nn #1 #2
2242 {
2243   \str_set_eq:NN \l_tmpa_str \l_keys_key_str
2244   \str_replace_all:Nnn \l_tmpa_str { ~ } { - }
2245   \str_set:Ne \l_tmpa_str { \str_lowercase:f { \l_tmpa_str } }
2246   \bool_set_false:N \l_tmpa_bool
2247   \clist_map_inline:nn { #1 }
2248   {
2249     \keys_if_exist:neT { ##1 } { \l_tmpa_str }
2250     {
```

```

2251         \@@_error:n { key-with-normal-form-exists }
2252         \bool_set_true:N \l_tmpa_bool
2253         \clist_map_break:
2254     }
2255 }
2256 \bool_if:NF \l_tmpa_bool { \@@_error:n { #2 } }
2257 }
2258 \@@_msg_new:nn { key-with-normal-form-exists }
2259 {
2260     The-key~'\l_keys_key_str'~does~not~exists.~It~will~be~ignored.\\
2261     Maybe~you~want~to~use~the~key~'\l_tmpa_str'.
2262 }
2263 \@@_msg_new:nn { No-environment-piton }
2264 {
2265     There~is~no~environment~piton!\\
2266     There~is~an~environment~{Piton}~and~a~command~
2267     \token_to_str:N \piton\ but~there~is~no~environment~
2268     {piton}.
2269 }
2270 \@@_msg_new:nn { rounded-corners-without-Tikz }
2271 {
2272     TikZ~not~used \\
2273     You~can't~use~the~key~'rounded-corners'~because~
2274     you~have~not~loaded~the~package~TikZ.~
2275     If~you~go~on,~your~key~will~be~ignored.~
2276     You~won't~have~similar~error~till~the~end~of~the~document.
2277 }
2278 \@@_msg_new:nn { tcolorbox-not-loaded }
2279 {
2280     tcolorbox~not~loaded \\
2281     You~can't~use~the~key~'tcolorbox'~because~
2282     you~have~not~loaded~the~package~tcolorbox.~
2283     Use~\token_to_str:N \usepackage[breakable]{tcolorbox}.~
2284     If~you~go~on,~that~key~will~be~ignored.
2285 }
2286 \@@_msg_new:nn { library-breakable-not-loaded }
2287 {
2288     breakable~not~loaded \\
2289     You~can't~use~the~key~'tcolorbox'~because~
2290     you~have~not~loaded~the~library~'breakable'~of~tcolorbox'.~
2291     You~should~load~'piton'~with~the~key~'breakable'~or~
2292     use~\token_to_str:N \tcbuselibrary{breakable}~
2293     in~the~preamble~of~your~document.~
2294     If~you~go~on,~that~key~will~be~ignored.
2295 }
2296 \@@_msg_new:nn { Language-not-defined }
2297 {
2298     Language~not~defined \\
2299     The~language~'\l_tmpa_tl'~has~not~been~defined~previously.~
2300     If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2301     will~be~ignored.
2302 }
2303 \@@_msg_new:nn { bad-version-of-piton.lua }
2304 {
2305     Bad~number~version~of~'piton.lua'\\
2306     The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2307     version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2308     address~that~issue.
2309 }
2310 \@@_msg_new:nn { Unknown-key-NewPitonLanguage }

```

```

2311 {
2312     Unknown~key~for~\token_to_str:N \NewPitonLanguage.\\
2313     The~key~'\l_keys_key_str'~is~unknown.~
2314     This~key~will~be~ignored.
2315 }
2316 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
2317 {
2318     The~style~'\l_keys_key_str'~is~unknown.\\
2319     This~setting~will~be~ignored.~
2320     The~available~styles~are~(in~alphabetic~order):~
2321     \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2322 }
2323 \@@_msg_new:nn { Invalid~key }
2324 {
2325     Wrong~use~of~key.\\
2326     You~can't~use~the~key~'\l_keys_key_str'~here.~
2327     Your~key~will~be~ignored.
2328 }
2329 \@@_msg_new:nn { Unknown~key~for~line~numbers }
2330 {
2331     Unknown~key. \\
2332     The~key~'line~numbers / \l_keys_key_str'~is~unknown.~
2333     The~available~keys~of~the~family~'line~numbers'~are~(in~
2334     alphabetic~order):~
2335     absolute,~false,~format,~label~empty~lines,~lmonoid10~drawn,~
2336     ~position,~resume,~skip~empty~lines,~sep,~start~and~true.~
2337     Your~key~will~be~ignored.
2338 }
2339 \@@_msg_new:nn { Unknown~key~for~marker }
2340 {
2341     Unknown~key. \\
2342     The~key~'marker / \l_keys_key_str'~is~unknown.~
2343     The~available~keys~of~the~family~'marker'~are~(in~
2344     alphabetic~order):~ beginning,~end~and~include~lines.~
2345     Your~key~will~be~ignored.
2346 }
2347 \@@_msg_new:nn { bad~range~specification }
2348 {
2349     Incompatible~keys.\\
2350     You~can't~specify~the~range~of~lines~to~include~by~using~both~
2351     markers~and~explicit~number~of~lines.~
2352     Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2353 }
2354 \cs_new_nopar:Nn \@@_thepage:
2355 {
2356     \thepage
2357     \cs_if_exist:NT \insertframenumber
2358     {
2359         ~(frame~\insertframenumber
2360         \cs_if_exist:NT \beamer@slidenum { ,~slide~\insertslidenum }
2361         )
2362     }
2363 }

```

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`.

```

2364 \@@_msg_new:nn { SyntaxError }
2365 {
2366     Syntax~Error~on~page~\@@_thepage:.\\
2367     Your~code~of~the~language~'\l_piton_language_str'~is~not~

```

```

2368 syntactically~correct.~
2369 It~won't~be~printed~in~the~PDF~file.
2370 }

2371 \@@_msg_new:nn { FileError }
2372 {
2373   File~Error.\\
2374   It's~not~possible~to~write~on~the~file~'#1' \\
2375   \sys_if_shell_unrestricted:F
2376   { (try~to~compile~with~'lua~l~atex~--shell~escape').\\ }
2377   If~you~go~on,~nothing~will~be~written~on~that~file.
2378 }

2379 \@@_msg_new:nn { InexistentDirectory }
2380 {
2381   Inexistent~directory.\\
2382   The~directory~'\l_@@_path_write_str'~
2383   given~in~the~key~'path~write'~does~not~exist.~
2384   Nothing~will~be~written~on~'\l_@@_write_str'.
2385 }

2386 \@@_msg_new:nn { begin~marker~not~found }
2387 {
2388   Marker~not~found.\\
2389   The~range~'\l_@@_begin_range_str'~provided~to~the~
2390   command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
2391   The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
2392 }

2393 \@@_msg_new:nn { end~marker~not~found }
2394 {
2395   Marker~not~found.\\
2396   The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
2397   provided~to~the~command~\token_to_str:N \PitonInputFile\
2398   has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
2399   be~inserted~till~the~end.
2400 }

2401 \@@_msg_new:nn { Unknown~file }
2402 {
2403   Unknown~file. \\
2404   The~file~'#1'~is~unknown.~
2405   Your~command~\token_to_str:N \PitonInputFile\ will~be~ignored.
2406 }

2407 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2408 {
2409   \bool_if:NF \g_@@_beamer_bool
2410   { \@@_error_or_warning:n { Without~beamer } }
2411 }

2412 \@@_msg_new:nn { Without~beamer }
2413 {
2414   Key~'\l_keys_key_str'~without~Beamer.\\
2415   You~should~not~use~the~key~'\l_keys_key_str'~since~you~
2416   are~not~in~Beamer.~However,~you~can~go~on.
2417 }

2418 \@@_msg_new:nn { rowcolor~in~detected~commands }
2419 {
2420   'rowcolor'~forbidden~in~'detected~commands'.\\
2421   You~should~put~'rowcolor'~in~'raw~detected~commands'.~
2422   That~key~will~be~ignored.
2423 }

2424 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2425 {
2426   Unknown~key. \\
2427   The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~

```

```

2428   It~will~be~ignored.~
2429   For~a~list~of~the~available~keys,~type~H~<return>.
2430 }
2431 {
2432   The~available~keys~are~(in~alphabetic~order):~
2433   annotation,~
2434   add-to-split-separation,~
2435   after-before-escape,~
2436   auto-gobble,~
2437   background-color,~
2438   before-end-escape,~
2439   begin-range,~
2440   box,~
2441   break-lines,~
2442   break-lines-in-piton,~
2443   break-lines-in-Piton,~
2444   break-numbers-anywhere,~
2445   break-strings-anywhere,~
2446   continuation-symbol,~
2447   continuation-symbol-on-indentation,~
2448   detected-beamer-commands,~
2449   detected-beamer-environments,~
2450   detected-commands,~
2451   end-of-broken-line,~
2452   end-range,~
2453   env-gobble,~
2454   env-used-by-split,~
2455   font-command(+),~
2456   gobble,~
2457   indent-broken-lines,~
2458   join,~
2459   label-as-zlabel,~
2460   language,~
2461   left-margin,~
2462   line-numbers/,~
2463   marker/,~
2464   math-comments,~
2465   no-join,~
2466   no-write,~
2467   path,~
2468   path-write,~
2469   print,~
2470   prompt-background-color,~
2471   raw-detected-commands,~
2472   resume,~
2473   right-margin,~
2474   rounded-corners,~
2475   show-spaces,~
2476   show-spaces-in-strings,~
2477   splittable,~
2478   splittable-on-empty-lines,~
2479   split-on-empty-lines,~
2480   split-separation,~
2481   tabs-auto-gobble,~
2482   tab-size,~
2483   tcolorbox,~
2484   varwidth,~
2485   vertical-detected-commands,~
2486   width~and~write.
2487 }

2488 \@@_msg_new:nn { label-with-lines-numbers }
2489 {

```

```

2490   You~can't~use~the~command~\token_to_str:N \label\
2491   or~\token_to_str:N \zlabel\
2492   because~the~key~'line-numbers'~
2493   is~not~active.~If~you~go~on,~that~command~will~ignored.
2494 }

2495 \@@_msg_new:nn { overlay~without~beamer }
2496 {
2497   You~can't~use~an~argument~<...>~for~your~command~
2498   \token_to_str:N \PitonInputFile\ because~you~are~not~
2499   in~Beamer.~If~you~go~on,~that~argument~will~be~ignored.
2500 }

2501 \@@_msg_new:nn { label~as~zlabel~needs~zref~package }
2502 {
2503   The~key~'label-as-zlabel'~requires~the~package~'zref'.~
2504   Please~load~the~package~'zref'~before~setting~the~key.
2505 }
2506 \hook_gput_code:nnn { begindocument } { . }
2507 {
2508   \bool_if:NT \g_@@_label_as_zlabel_bool
2509   {
2510     \IfPackageLoadedF { zref-base }
2511     { \@@_fatal:n { label~as~zlabel~needs~zref~package } }
2512   }
2513 }

```

2.15 The glyphs of the 10 arabic numbers

```

2514 \cs_new_protected:Npn \@@_draw_number:n #1
2515 { \tl_map_inline:nn { #1 } { \use:c { c_@@_glyph_ ##1 _tl } } }
2516 \cs_generate_variant:Nn \@@_draw_number:n { e }

2517 \cs_set_eq:NN \tlconstcn \tl_const:cn

2518 \ExplSyntaxOff
2519 \tlconstcn{c_@@_glyph_0_tl}
2520 {%
2521   \hbox to 5pt
2522   {%
2523     \hfill
2524     \pdfextension literal
2525     {
2526       4.24 3.05 m
2527       4.24 4.91 3.22 6.22 2.12 6.22 c
2528       1.00 6.22 0 4.88 0 3.06 c
2529       0 1.20 1.02 -0.11 2.12 -0.11 c
2530       3.24 -0.11 4.24 1.23 4.24 3.05 c
2531       3.55 3.16 m
2532       3.55 1.68 2.90 0.50 2.12 0.50 c
2533       1.34 0.50 0.69 1.68 0.69 3.16 c
2534       0.69 4.62 1.38 5.61 2.12 5.61 c
2535       2.85 5.61 3.55 4.63 3.55 3.16 c
2536     }
2537     }%
2538   \hfill
2539   }%
2540 }

2541 \tlconstcn{c_@@_glyph_1_tl}
2542 {%
2543   \hbox to 5pt

```

```

2544     {%
2545         \hfill \kern 1 pt
2546         \pdfextension literal
2547         {
2548             3.37 0.3 m
2549             3.37 0.61 3.13 0.61 2.97 0.61 c
2550             2.06 0.61 l
2551             2.06 5.81 l
2552             2.06 5.97 2.06 6.22 1.76 6.22 c
2553             1.57 6.22 1.51 6.1 1.46 5.98 c
2554             1.08 5.13 0.56 5.02 0.37 5.0 c
2555             0.21 4.99 0.0 4.97 0.0 4.69 c
2556             0.0 4.44 0.18 4.39 0.33 4.39 c
2557             0.52 4.39 0.93 4.45 1.37 4.83 c
2558             1.37 0.61 l
2559             0.46 0.61 l
2560             0.3 0.61 0.06 0.61 0.06 0.3 c
2561             0.06 0.0 0.31 0.0 0.46 0.0 c
2562             2.97 0.0 l
2563             3.12 0.0 3.37 0.0 3.37 0.3 c
2564             h f
2565         }%
2566     \hfill
2567 }%
2568 }

2569 \tlconstcn{c_@@_glyph_2_tl}
2570 {%
2571     \hbox to 5pt
2572     {%
2573         \hfill
2574         \pdfextension literal
2575         {
2576             4.2 0.41 m
2577             4.2 0.67 l
2578             4.2 0.86 4.2 1.08 3.86 1.08 c
2579             3.51 1.08 3.51 0.89 3.51 0.61 c
2580             1.13 0.61 l
2581             1.72 1.12 2.68 1.87 3.11 2.27 c
2582             3.74 2.83 4.2 3.47 4.2 4.27 c
2583             4.2 5.47 3.19 6.22 1.97 6.22 c
2584             0.79 6.22 0.0 5.4 0.0 4.55 c
2585             0.0 4.18 0.28 4.07 0.45 4.07 c
2586             0.66 4.07 0.89 4.24 0.89 4.52 c
2587             0.89 4.64 0.84 4.77 0.75 4.84 c
2588             0.9 5.3 1.37 5.61 1.92 5.61 c
2589             2.74 5.61 3.51 5.15 3.51 4.27 c
2590             3.51 3.57 3.02 2.99 2.36 2.44 c
2591             0.15 0.58 l
2592             0.06 0.5 0.0 0.45 0.0 0.31 c
2593             0.0 0.0 0.25 0.0 0.41 0.0 c
2594             3.8 0.0 l
2595             4.13 0.0 4.2 0.09 4.2 0.41 c
2596             h f
2597         }%
2598     \hfill
2599 }%
2600 }

2601 \tlconstcn{c_@@_glyph_3_tl}
2602 {%
2603     \hbox to 5pt
2604     {%
2605         \hfill
2606         \pdfextension literal

```



```

2607 {
2608     4.36 1.74 m
2609     4.36 2.45 3.89 3.04 3.23 3.34 c
2610     3.79 3.7 4.07 4.27 4.07 4.81 c
2611     4.07 5.55 3.34 6.22 2.19 6.22 c
2612     0.99 6.22 0.29 5.74 0.29 5.05 c
2613     0.29 4.72 0.54 4.58 0.74 4.58 c
2614     0.95 4.58 1.18 4.75 1.18 5.03 c
2615     1.18 5.17 1.12 5.27 1.09 5.3 c
2616     1.4 5.61 2.11 5.61 2.2 5.61 c
2617     2.88 5.61 3.38 5.25 3.38 4.8 c
2618     3.38 4.5 3.23 4.15 2.96 3.93 c
2619     2.64 3.67 2.39 3.65 2.03 3.63 c
2620     1.46 3.59 1.31 3.59 1.31 3.3 c
2621     1.31 2.99 1.55 2.99 1.71 2.99 c
2622     2.17 2.99 l
2623     3.16 2.99 3.67 2.32 3.67 1.74 c
2624     3.67 1.13 3.11 0.5 2.2 0.5 c
2625     1.8 0.5 1.03 0.61 0.77 1.08 c
2626     0.82 1.13 0.89 1.19 0.89 1.39 c
2627     0.89 1.63 0.7 1.83 0.45 1.83 c
2628     0.22 1.83 0.0 1.68 0.0 1.36 c
2629     0.0 0.47 0.97 -0.11 2.2 -0.11 c
2630     3.51 -0.11 4.36 0.81 4.36 1.74 c
2631     h f
2632 }%
2633 \hfill
2634 }%
2635 }
2636 \tlconstcn{c_@@_glyph_4_tl}
2637 {%
2638     \hbox to 5pt
2639     {%
2640         \hfill
2641         \pdfextension literal
2642         {
2643             4.66 1.99 m
2644             4.66 2.3 4.42 2.3 4.26 2.3 c
2645             3.48 2.3 l
2646             3.48 5.82 l
2647             3.48 6.15 3.41 6.23 3.07 6.23 c
2648             2.79 6.23 l
2649             2.55 6.23 2.5 6.22 2.37 6.02 c
2650             0.09 2.44 l
2651             0.0 2.31 0.0 2.29 0.0 2.09 c
2652             0.0 1.76 0.09 1.69 0.4 1.69 c
2653             2.92 1.69 l
2654             2.92 0.61 l
2655             2.3 0.61 l
2656             2.14 0.61 1.9 0.61 1.9 0.3 c
2657             1.9 0.0 2.15 0.0 2.3 0.0 c
2658             4.1 0.0 l
2659             4.25 0.0 4.5 0.0 4.5 0.3 c
2660             4.5 0.61 4.26 0.61 4.1 0.61 c
2661             3.48 0.61 l
2662             3.48 1.69 l
2663             4.26 1.69 l
2664             4.41 1.69 4.66 1.69 4.66 1.99 c
2665             2.92 2.3 m
2666             0.7 2.3 l
2667             2.92 5.79 l
2668             h f
2669         }%

```

```

2670     \hfill
2671   }%
2672 }
2673 \tlconstcn{c_@@_glyph_5_tl}
2674 {%
2675   \hbox to 5pt
2676   {%
2677     \hfill
2678     \pdfextension literal
2679     {
2680       4.2 1.9 m
2681       4.2 2.91 3.43 3.89 2.25 3.89 c
2682       1.9 3.89 1.46 3.82 1.05 3.6 c
2683       1.05 5.5 l
2684       3.45 5.5 l
2685       3.6 5.5 3.85 5.5 3.85 5.8 c
2686       3.85 6.11 3.61 6.11 3.45 6.11 c
2687       0.76 6.11 l
2688       0.43 6.11 0.36 6.02 0.36 5.7 c
2689       0.36 3.04 l
2690       0.36 2.86 0.36 2.63 0.68 2.63 c
2691       0.86 2.63 0.9 2.68 0.98 2.78 c
2692       1.25 3.1 1.66 3.28 2.24 3.28 c
2693       3.06 3.28 3.51 2.55 3.51 1.9 c
2694       3.51 1.1 2.8 0.5 1.96 0.5 c
2695       1.68 0.5 1.04 0.58 0.76 1.13 c
2696       0.81 1.18 0.89 1.25 0.89 1.45 c
2697       0.89 1.74 0.66 1.9 0.45 1.9 c
2698       0.3 1.9 0.0 1.81 0.0 1.42 c
2699       0.0 0.59 0.84 -0.11 1.96 -0.11 c
2700       3.21 -0.11 4.2 0.79 4.2 1.9 c
2701     h f
2702   }%
2703   \hfill
2704 }%
2705 }
2706 \tlconstcn{c_@@_glyph_6_tl}
2707 {%
2708   \hbox to 5pt
2709   {%
2710     \hfill
2711     \pdfextension literal
2712     {
2713       4.18 1.93 m
2714       4.18 3.07 3.3 3.96 2.2 3.96 c
2715       1.68 3.96 1.16 3.79 0.71 3.38 c
2716       0.85 4.79 1.79 5.61 2.67 5.61 c
2717       3.01 5.61 3.17 5.5 3.22 5.45 c
2718       3.17 5.4 3.12 5.34 3.12 5.16 c
2719       3.12 4.92 3.3 4.72 3.56 4.72 c
2720       3.81 4.72 4.01 4.89 4.01 5.19 c
2721       4.01 5.68 3.65 6.22 2.68 6.22 c
2722       1.34 6.22 0.0 5.01 0.0 2.99 c
2723       0.0 0.62 1.12 -0.11 2.11 -0.11 c
2724       3.2 -0.11 4.18 0.73 4.18 1.93 c
2725       3.49 1.93 m
2726       3.49 1.07 2.83 0.5 2.11 0.5 c
2727       1.46 0.5 1.07 0.99 0.87 1.6 c
2728       0.77 1.84 0.79 2.08 0.79 2.22 c
2729       0.79 2.84 1.39 3.34 2.15 3.34 c
2730       2.96 3.34 3.49 2.67 3.49 1.93 c
2731     h f
2732   }%

```

```

2733     \hfill
2734 }%
2735 }
2736 \tlconstcn{c_@@_glyph_7_tl}
2737 {%
2738   \hbox to 5pt
2739   {%
2740     \hfill
2741     \pdfextension literal
2742     {
2743       4.36 5.8 m
2744       4.36 6.11 4.12 6.11 3.96 6.11 c
2745       0.66 6.11 l
2746       0.6 6.27 0.42 6.27 0.34 6.27 c
2747       0.0 6.27 0.0 6.05 0.0 5.86 c
2748       0.0 5.44 l
2749       0.0 5.25 0.0 5.03 0.34 5.03 c
2750       0.69 5.03 0.69 5.22 0.69 5.5 c
2751       3.33 5.5 l
2752       1.6 3.66 1.26 1.46 1.26 0.36 c
2753       1.26 0.22 1.26 -0.11 1.61 -0.11 c
2754       1.78 -0.11 1.95 0.0 1.95 0.29 c
2755       2.0 3.13 3.53 4.87 4.14 5.46 c
2756       4.34 5.64 4.36 5.66 4.36 5.8 c
2757     h f
2758   }%
2759   \hfill
2760 }%
2761 }
2762 \tlconstcn{c_@@_glyph_8_tl}
2763 {%
2764   \hbox to 5pt
2765   {%
2766     \hfill
2767     \pdfextension literal
2768     {
2769       4.36 1.74 m
2770       4.36 2.49 3.71 3.08 2.97 3.29 c
2771       3.77 3.56 4.22 4.05 4.22 4.62 c
2772       4.22 5.44 3.37 6.22 2.18 6.22 c
2773       0.98 6.22 0.14 5.43 0.14 4.62 c
2774       0.14 4.05 0.6 3.55 1.39 3.29 c
2775       0.64 3.08 0.0 2.49 0.0 1.74 c
2776       0.0 0.77 0.93 -0.11 2.18 -0.11 c
2777       3.44 -0.11 4.36 0.77 4.36 1.74 c
2778       3.53 4.61 m
2779       3.53 4.04 2.91 3.6 2.18 3.6 c
2780       1.45 3.6 0.83 4.05 0.83 4.61 c
2781       0.83 5.13 1.39 5.61 2.18 5.61 c
2782       2.96 5.61 3.53 5.13 3.53 4.61 c
2783       3.67 1.75 m
2784       3.67 1.06 3.01 0.5 2.18 0.5 c
2785       1.35 0.5 0.69 1.06 0.69 1.75 c
2786       0.69 2.36 1.27 2.99 2.18 2.99 c
2787       3.1 2.99 3.67 2.36 3.67 1.75 c
2788     h f
2789   }%
2790   \hfill
2791 }%
2792 }
2793 \tlconstcn{c_@@_glyph_9_tl}
2794 {%
2795   \hbox to 5pt

```

```

2796     {%
2797         \hfill
2798         \pdfextension literal
2799         {
2800             4.18 3.12 m
2801             4.18 5.53 3.07 6.22 2.12 6.22 c
2802             1.01 6.22 0.0 5.39 0.0 4.18 c
2803             0.0 3.04 0.88 2.15 1.98 2.15 c
2804             2.5 2.15 3.02 2.32 3.47 2.73 c
2805             3.37 1.49 2.59 0.5 1.63 0.5 c
2806             1.54 0.5 1.18 0.51 0.97 0.67 c
2807             1.01 0.72 1.06 0.78 1.06 0.95 c
2808             1.06 1.19 0.88 1.39 0.62 1.39 c
2809             0.37 1.39 0.17 1.22 0.17 0.92 c
2810             0.17 0.6 0.35 -0.11 1.63 -0.11 c
2811             2.95 -0.11 4.18 1.14 4.18 3.12 c
2812             3.4 3.89 m
2813             3.4 3.34 2.86 2.77 2.03 2.77 c
2814             1.22 2.77 0.69 3.44 0.69 4.18 c
2815             0.69 5.05 1.4 5.61 2.12 5.61 c
2816             2.55 5.61 2.83 5.38 2.99 5.18 c
2817             3.31 4.79 3.4 4.6 3.4 3.89 c
2818             h f
2819         }%
2820     \hfill
2821 }%
2822 }
2823 \ExplSyntaxOn

```

2.16 We load piton.lua

```

2824 \cs_new_protected:Npn \@@_test_version:n #1
2825 {
2826     \str_if_eq:onF \PitonFileVersion { #1 }
2827     { \@@_error:n { bad~version~of~piton.lua } }
2828 }

2829 \hook_gput_code:nnn { begindocument } { . }
2830 {
2831     \lua_load_module:n { piton }
2832     \lua_now:n
2833     {
2834         tex.sprint ( luatexbase.catcodetables.expl ,
2835                     [[\@@_test_version:n {}] .. piton_version .. "]" )
2836     }
2837 }
</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`.

```

2838 ⟨*LUA⟩
2839 piton.comment_latex = piton.comment_latex or ">"
2840 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`.

```

2841 piton.write_files = { }
2842 piton.join_files = { }

2843 local sprintL3
2844 function sprintL3 ( s )
2845     tex.sprint ( luatexbase.catcodetables.expl , s )
2846 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.

```

2847 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2848 local Cg , Cmt , Cb = lpeg.Cg , lpeg.Cmt , lpeg.Cb
2849 local B , R = lpeg.B , lpeg.R

```

The following line is mandatory.

```

2850 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”).

```

2851 local Q
2852 function Q ( pattern )
2853     return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2854 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.

```

2855 local L
2856 function L ( pattern ) return
2857     Ct ( C ( pattern ) )
2858 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.

```

2859 local Lc
2860 function Lc ( string ) return
2861     Cc ( { luatexbase.catcodetables.expl , string } )
2862 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)

```

2863 local K
2864 function K ( style , pattern ) return
2865   Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
2866   * Q ( pattern )
2867   * Lc "}}"
2868 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.

```

2869 local WithStyle
2870 function WithStyle ( style , pattern ) return
2871   Ct ( Cc "Open" * Cc ( [[{\PitonStyle{ ]] .. style .. "}{" ) * Cc "}}" )
2872   * pattern
2873   * Ct ( Cc "Close" )
2874 end

```

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

```

2875 Escape = P ( false )
2876 EscapeClean = P ( false )
2877 if piton.begin_escape then
2878   Escape =
2879     P ( piton.begin_escape )
2880     * Lc [[\l_@@_after_begin_escape_tl]]
2881     * L ( ( 1 - P ( piton.end_escape ) ) ^ 0 )
2882     * Lc [[\l_@@_before_end_escape_tl]]
2883     * 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).

```

2884 EscapeClean =
2885   P ( piton.begin_escape )
2886   * ( 1 - P ( piton.end_escape ) ) ^ 0
2887   * P ( piton.end_escape )
2888 end
2889 EscapeMath = P ( false )
2890 if piton.begin_escape_math then
2891   EscapeMath =
2892     P ( piton.begin_escape_math )
2893     * Lc "$"
2894     * L ( ( 1 - P ( piton.end_escape_math ) ) ^ 1 )
2895     * Lc "$"
2896     * P ( piton.end_escape_math )
2897 end

```

The basic syntactic LPEG

```

2898 local alpha , digit = lpeg.alpha , lpeg.digit
2899 local space = P " "

```

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

```

2900 local letter = alpha + "_" + "â" + "ã" + "ç" + "ê" + "è" + "é" + "ë" + "ï" + "î"
2901             + "ô" + "û" + "ü" + "Â" + "Ã" + "Ç" + "É" + "Ê" + "Ë" + "Ï"
2902             + "Î" + "Ï" + "Ô" + "Õ" + "Û"
2903
2904 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).

```

2905 local identifier = letter * alphanum ^ 0

```

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

```

2906 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

```

2907 local allow_underscores_except_first
2908 function allow_underscores_except_first ( p )
2909     return p * ( P "_" + p ) ^ 0
2910 end
2911 local allow_underscores
2912 function allow_underscores ( p )
2913     return ( P "_" + p ) ^ 0
2914 end
2915 local digits_to_number
2916 function digits_to_number(prefix, digits)
2917     -- The edge cases of what is allowed in number literals is modelled after
2918     -- OCaml numbers, which seems to be the most permissive language
2919     -- in this regard (among C, OCaml, Python & SQL).
2920     return prefix
2921         * allow_underscores_except_first(digits^1)
2922         * ( P "." * #(1 - P ".") * allow_underscores(digits) ) ^ -1
2923         * ( S "eE" * S "+-" ^ -1 * allow_underscores_except_first(digits^1) ) ^ -1
2924 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).

```

2925 local Number =
2926     K ( 'Number.Internal' ,
2927         digits_to_number ( P "0x" + P "0X", R "af" + R "AF" + digit )
2928         + digits_to_number ( P "0o" + P "0O", R "07" )
2929         + digits_to_number ( P "0b" + P "0B", R "01" )
2930         + digits_to_number ( "" , digit )
2931     )

```

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?).

```

2932 local lpeg_central = 1 - S " '\r[({}])'" - digit

```

³The edge cases such as

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

```

2933 if piton.begin_escape then
2934   lpeg_central = lpeg_central - piton.begin_escape
2935 end
2936 if piton.begin_escape_math then
2937   lpeg_central = lpeg_central - piton.begin_escape_math
2938 end
2939 local Word = Q ( lpeg_central ^ 1 )

2940 local Space = Q " " ^ 1
2941 local SkipSpace = Q " " ^ 0
2942
2943 local Punct = Q ( S ".,:;!" )
2944
2945 local Tab = "\t" * Lc [[ \@@_tab: ]]

2946 local LeadingSpace = Lc [[ \@@_leading_space: ]] * P " "

2947 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.

```

2948 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).

```

2949 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
2950 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
2951 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
2952 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.

```

2953 local detectedCommands = P ( false )
2954 for _ , x in ipairs ( detected_commands ) do
2955   detectedCommands = detectedCommands + P ( "\\" .. x )
2956 end

```

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

```

2957 local rawDetectedCommands = P ( false )
2958 for _ , x in ipairs ( raw_detected_commands ) do
2959   rawDetectedCommands = rawDetectedCommands + P ( "\\" .. x )
2960 end

```



```

2961 local beamerCommands = P ( false )
2962 for _ , x in ipairs ( beamer_commands ) do
2963   beamerCommands = beamerCommands + P ( "\\\" .. x )
2964 end
2965
2965 local beamerEnvironments = P ( false )
2966 for _ , x in ipairs ( beamer_environments ) do
2967   beamerEnvironments = beamerEnvironments + P ( x )
2968 end

```

Several tools for the construction of the main LPEG

```

2969 local LPEG0 = { }
2970 local LPEG1 = { }
2971 local LPEG2 = { }
2972 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*.

```

2973 local Compute_braces
2974 function Compute_braces ( lpeg_string ) return
2975   P { "E" ,
2976     E =
2977     (
2978       "{" * V "E" * "}"
2979       +
2980       lpeg_string
2981       +
2982       ( 1 - S "{" )
2983     ) ^ 0
2984   }
2985 end

```

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

```

2986 local Compute_DetectedCommands
2987 function Compute_DetectedCommands ( lang , braces ) return
2988   Ct (
2989     Cc "Open"
2990     * C ( detectedCommands * space ^ 0 * P "{" )
2991     * Cc "}"
2992   )
2993   * ( braces
2994     / ( function ( s )
2995         if s ~= '' then return
2996           LPEG1[lang] : match ( s )
2997         end
2998       end )
2999   )
3000   * P "}"
3001   * Ct ( Cc "Close" )
3002 end
3003
3003 local Compute_RawDetectedCommands
3004 function Compute_RawDetectedCommands ( lang , braces ) return
3005   Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
3006 end

```

```

3007 local Compute_LPEG_cleaner
3008 function Compute_LPEG_cleaner ( lang , braces ) return
3009   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
3010         * ( braces
3011           / ( function ( s )
3012               if s ~= '' then return
3013                 LPEG_cleaner[lang] : match ( s )
3014             end
3015           end )
3016         )
3017       * "}"
3018     + EscapeClean
3019     + C ( P ( 1 ) )
3020   ) ^ 0 ) / table.concat
3021 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`.

```

3022 local ParseAgain
3023 function ParseAgain ( code )
3024   if code ~= '' then return

```

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

```

3025   LPEG1[piton.language] : match ( code )
3026 end
3027 end

```

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

```

3028 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.

```

3029 local Compute_Beamer
3030 function Compute_Beamer ( lang , braces )

```

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

```

3031 local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
3032 lpeg = lpeg +
3033   Ct ( Cc "Open"
3034         * C ( beamerCommands
3035               * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3036               * P "{"
3037             )
3038         * Cc "}"
3039       )
3040   * ( braces /
3041       ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3042   * "}"
3043   * Ct ( Cc "Close" )

```

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

```

3044 lpeg = lpeg +
3045   L ( P [[\alt]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
3046   * ( braces /
3047     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3048   * L ( P "}" )
3049   * ( braces /
3050     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3051   * L ( P "}" )

```

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

```

3052 lpeg = lpeg +
3053   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
3054   * ( braces
3055     / ( function ( s )
3056         if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3057   * L ( P "}" )
3058   * ( braces
3059     / ( function ( s )
3060         if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3061   * L ( P "}" )
3062   * ( braces
3063     / ( function ( s )
3064         if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3065   * L ( P "}" )

```

Now, the environments of Beamer.

```

3066 for _ , x in ipairs ( beamer_environments ) do
3067   lpeg = lpeg +
3068     Ct ( Cc "Open"
3069       * C (
3070         P ( [[\begin{]] .. x .. "]" )
3071         * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3072       )
3073       * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
3074       * Cc ( [[\end{]] .. x .. "]" )
3075     )
3076   * (

```

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).

```

3077     (
3078       P { "E" ,
3079         E = (
3080           P ( [[\begin{]] .. x .. "]" )
3081           * V "E"
3082           * P ( [[\end{]] .. x .. "]" )
3083         +
3084         (
3085           1
3086           - P ( [[\begin{]] .. x .. "]" )
3087           - P ( [[\end{]] .. x .. "]" )
3088         )
3089       ) ^ 0
3090     }
3091   )
3092   / ( function ( s )
3093     if s ~= '' then return
3094       LPEG1[lang] : match ( s )
3095     end
3096   end )
3097 )

```

```

3098         * P ( [[\end{}} .. x .. "}" )
3099         * Ct ( Cc "Close" )
3100     end

```

Now, you can return the value we have computed.

```

3101     return lpeg
3102 end

```

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

```

3103 local CommentMath =
3104     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`.

```

3105 local Prompt =
3106     K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
3107     * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

3108 local EOL =
3109     P "\r"
3110     *
3111     (
3112         space ^ 0 * -1
3113         +
3114         Cc "EOL"
3115     )
3116     * ( LeadingSpace ^ 0 * # ( 1 - S " \r" ) ) ^ -1

```

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

```

3117 local CommentLaTeX =
3118     P ( piton.comment_latex )
3119     * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}}]
3120     * L ( ( 1 - P "\r" ) ^ 0 )
3121     * Lc "}}"
3122     * ( 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).

```

3123 --python Python
3124 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.

```

3125 local Operator =
3126     K ( 'Operator' ,
3127         P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "://" + "*"
3128         + S "--+/*%=<>&.@|" )
3129
3130 local OperatorWord =
3131     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.

```

3132 local For = K ( 'Keyword' , P "for" )
3133         * Space
3134         * Identifier
3135         * Space
3136         * K ( 'Keyword' , P "in" )
3137
3138 local Keyword =
3139     K ( 'Keyword' ,
3140         P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
3141         "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
3142         "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
3143         "try" + "while" + "with" + "yield" + "yield from" )
3144     + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
3145
3146 local Builtin =
3147     K ( 'Name.Builtin' ,
3148         P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
3149         "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
3150         "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
3151         "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
3152         "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
3153         "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
3154         + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
3155         "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
3156         "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
3157         "vars" + "zip" )
3158
3159 local Exception =
3160     K ( 'Exception' ,
3161         P "ArithmeticError" + "AssertionError" + "AttributeError" +
3162         "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
3163         "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
3164         "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
3165         "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
3166         "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
3167         "NotImplementedError" + "OSError" + "OverflowError" +
3168         "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
3169         "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
3170         "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
3171         + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
3172         "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
3173         "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
3174         "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
3175         "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
3176         "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
3177         "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
3178         "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
3179         "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
3180         "RecursionError" )
3181
3182 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.

```

3183 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:`

```

3184 local DefClass =
3185     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`

```

3186 local ImportAs =
3187     K ( 'Keyword' , "import" )
3188     * Space
3189     * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
3190     * (
3191         ( Space * K ( 'Keyword' , "as" ) * Space
3192           * K ( 'Name.Namespace' , identifier ) )
3193         +
3194         ( SkipSpace * Q "," * SkipSpace
3195           * K ( 'Name.Namespace' , identifier ) ) ^ 0
3196     )

```

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`

```

3197 local FromImport =
3198     K ( 'Keyword' , "from" )
3199     * Space * K ( 'Name.Namespace' , identifier )
3200     * 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:

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

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

```

3201 local PercentInterpol =
3202     K ( 'String.Interpol' ,
3203         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`.

```

3204 * ( "(" * alphanum ^ 1 * ")" ) ^ -1
3205 * ( S "-#0 +" ) ^ 0
3206 * ( digit ^ 1 + "*" ) ^ -1
3207 * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
3208 * ( S "HLL" ) ^ -1
3209 * S "sdfFeExXorgiGauc%"
3210 )

```

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

```

3211 local SingleShortString =
3212   WithStyle ( 'String.Short.Internal' ,

```

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

```

3213   Q ( P "f'" + "F'" )
3214   * (
3215     K ( 'String.Interpol' , "{" )
3216     * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0 )
3217     * Q ( P ":" * ( 1 - S "}':" ) ^ 0 ) ^ -1
3218     * K ( 'String.Interpol' , "}" )
3219     +
3220     SpaceInString
3221     +
3222     Q ( ( P "\\'" + "\\\\" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
3223   ) ^ 0
3224   * Q ""
3225   +

```

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

```

3226   Q ( P "'" + "r'" + "R'" )
3227   * ( Q ( ( P "\\'" + "\\\\" + 1 - S " 'r%" ) ^ 1 )
3228     + SpaceInString
3229     + PercentInterpol
3230     + Q "%"
3231   ) ^ 0
3232   * Q "" )
3233 local DoubleShortString =
3234   WithStyle ( 'String.Short.Internal' ,
3235     Q ( P "f\"" + "F\"" )
3236     * (
3237       K ( 'String.Interpol' , "{" )
3238       * K ( 'Interpol.Inside' , ( 1 - S "}\" )" ) ^ 0 )
3239       * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}\" )" ) ^ 0 ) ) ^ -1
3240       * K ( 'String.Interpol' , "}" )
3241       +
3242       SpaceInString
3243       +
3244       Q ( ( P "\\\"" + "\\\\" + "{{" + "}}" + 1 - S " {}\" )" ^ 1 )
3245     ) ^ 0
3246     * Q "\"
3247   +
3248     Q ( P "\" + "r\"" + "R\"" )
3249     * ( Q ( ( P "\\\"" + "\\\\" + 1 - S " \"r%" ) ^ 1 )
3250       + SpaceInString
3251       + PercentInterpol
3252       + Q "%"
3253     ) ^ 0
3254     * 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`.

```

3255
3256 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.

```

3257 local braces =
3258   Compute_braces
3259   (
3260     ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
3261     * ( P '\\\"' + 1 - S "\"" ) ^ 0 * "\""
3262   +
3263     ( P '\'' + 'r\'' + 'R\'' + 'f\'' + 'F\'' )
3264     * ( P '\\\'' + 1 - S '\'' ) ^ 0 * '\''
3265   )
3266
3267 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

Detected commands

```

3268 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
3269 + Compute_RawDetectedCommands ( 'python' , braces )

```

LPEG_cleaner

```

3270 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

The long strings

```

3271 local SingleLongString =
3272   WithStyle ( 'String.Long.Internal' ,
3273     ( Q ( S "fF" * P "'''" )
3274       * (
3275         K ( 'String.Interpol' , "{" )
3276         * K ( 'Interpol.Inside' , ( 1 - S "};\r" - "'''" ) ^ 0 )
3277         * Q ( P ":" * ( 1 - S "};\r" - "'''" ) ^ 0 ) ^ -1
3278         * K ( 'String.Interpol' , "}" )
3279       +
3280         Q ( ( 1 - P "'''" - S "{'}\r" ) ^ 1 )
3281       +
3282         EOL
3283     ) ^ 0
3284   +
3285     Q ( ( S "rR" ) ^ -1 * "'''" )
3286     * (
3287       Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
3288       +
3289       PercentInterpol
3290       +
3291       P "%"
3292       +
3293       EOL
3294     ) ^ 0
3295   )
3296   * Q "'''" )

```



```

3297 local DoubleLongString =
3298   WithStyle ( 'String.Long.Internal' ,
3299     (
3300       Q ( S "fF" * "\"\\\"" )
3301       * (
3302         K ( 'String.Interpol', "{ " )
3303         * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "\"\\\"" ) ^ 0 )
3304         * Q ( ":" * (1 - S "}:\\r" - "\"\\\"" ) ^ 0 ) ^ -1
3305         * K ( 'String.Interpol' , "}" )
3306         +
3307         Q ( ( 1 - S "{}\\r" - "\"\\\"" ) ^ 1 )
3308         +
3309         EOL
3310       ) ^ 0
3311     +
3312     Q ( S "rR" ^ -1 * "\"\\\"" )
3313     * (
3314       Q ( ( 1 - P "\"\\\"" - S "%\\r" ) ^ 1 )
3315       +
3316       PercentInterpol
3317       +
3318       P "%"
3319       +
3320       EOL
3321     ) ^ 0
3322   )
3323   * Q "\"\\\""
3324 )
3325 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`).

```

3326 local StringDoc =
3327   K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\\\"" )
3328   * ( K ( 'String.Doc.Internal' , (1 - P "\"\\\"" - "\\r" ) ^ 0 ) * EOL
3329     * Tab ^ 0
3330   ) ^ 0
3331   * 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.

```

3332 local Comment =
3333   WithStyle
3334   ( 'Comment.Internal' ,
3335     Q "#" * ( CommentMath + Q ( ( 1 - S "$\\r" ) ^ 1 ) ) ^ 0 -- $
3336   )
3337   * ( 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).

```

3338 local expression =
3339   P { "E" ,
3340     E = ( "''" * ( P "\\'" + 1 - S "'\\r" ) ^ 0 * "''"
3341       + "\"\" * ( P "\\\"" + 1 - S "\"\\r" ) ^ 0 * "\"\"
3342       + "{" * V "F" * "}"
3343       + "(" * V "F" * ")" )

```

```

3344         + "[" * V "F" * "]"
3345         + ( 1 - S "{ } ( [ ] \r , " ) ) ^ 0 ,
3346     F = (      "{" * V "F" * "}"
3347           + "(" * V "F" * ")"
3348           + "[" * V "F" * "]"
3349           + ( 1 - S "{ } ( [ ] \r \'"' " ) ) ^ 0
3350     }

```

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`.

```

3351     local Params =
3352     P { "E" ,
3353         E = ( V "F" * ( Q " , " * V "F" ) ^ 0 ) ^ -1 ,
3354         F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
3355           * ( Q ":" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3356           * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
3357     }

```

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`...

```

3358     local DefFunction =
3359     K ( 'Keyword' , "def" )
3360     * Space
3361     * K ( 'Name.Function.Internal' , identifier )
3362     * SkipSpace
3363     * Q "(" * Params * Q ")"
3364     * SkipSpace
3365     * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3366     * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
3367     * Q ":"
3368     * ( SkipSpace
3369       * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
3370       * Tab ^ 0
3371       * SkipSpace
3372       * StringDoc ^ 0 -- there may be additional docstrings
3373     ) ^ -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

```

3374     local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

The main LPEG for the language Python

```
3375 local EndKeyword
3376     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3377     EscapeMath + -1
```

First, the main loop :

```
3378 local Main =
3379     space ^ 0 * EOL
3380     + Space
3381     + Tab
3382     + Escape + EscapeMath
3383     + Beamer
3384     + CommentLaTeX
3385     + DetectedCommands
3386     + Prompt
3387     + LongString
3388     + Comment
3389     + ExceptionInConsole
3390     + Delim
3391     + Operator
3392     + OperatorWord * EndKeyword
3393     + ShortString
3394     + Punct
3395     + FromImport
3396     + RaiseException
3397     + DefFunction
3398     + DefClass
3399     + For
3400     + Keyword * EndKeyword
3401     + Decorator
3402     + Builtin * EndKeyword
3403     + Identifier
3404     + Number
3405     + Word
```

Here, we must not put local, of course.

```
3406 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.

```
3407 LPEG2.python =
3408     Ct (
3409         ( space ^ 0 * "\r" ) ^ -1
3410         * Lc [[ \@@_begin_line: ]]
3411         * LeadingSpace ^ 0
3412         * ( space ^ 1 * -1 + Main ) ^ 0
3413         * -1
3414         * Lc [[ \@@_end_line: ]]
3415     )
```

End of the Lua scope for the language Python.

```
3416 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).

```

3417 --ocaml Ocaml OCaml
3418 do

3419     local SkipSpace = ( Q " " + EOL ) ^ 0
3420     local Space = ( Q " " + EOL ) ^ 1

3421     local braces = Compute_braces ( '\'' * ( 1 - S "\"" ) ^ 0 * '\'' )

3422     if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
3423     DetectedCommands =
3424         Compute_DetectedCommands ( 'ocaml' , braces )
3425         + Compute_RawDetectedCommands ( 'ocaml' , braces )
3426     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.

```

3427     function Q ( pattern, strict )
3428         if strict ~= nil then
3429             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3430         else
3431             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3432                 + Beamer + DetectedCommands + EscapeMath + Escape
3433         end
3434     end

3435     local K
3436     function K ( style , pattern, strict ) return
3437         Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
3438         * Q ( pattern, strict )
3439         * Lc "}"
3440     end

3441     local WithStyle
3442     function WithStyle ( style , pattern ) return
3443         Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{ ]] .. style .. "}{" ) * Cc "}" )
3444         * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
3445         * Ct ( Cc "Close" )
3446     end

```

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

```

3447     local balanced_parens =
3448         P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }

```

The strings of OCaml

```

3449     local ocaml_string =
3450         P "\""
3451         * (
3452             P " "
3453             +
3454             P ( ( P '\\\'' + 1 - S " \r" ) ^ 1 )
3455             +
3456             EOL -- ?
3457         ) ^ 0
3458     * P "\""

```

```

3459 local String =
3460   WithStyle
3461     ( 'String.Long.Internal' ,
3462       Q "\""
3463       * (
3464         SpaceInString
3465         +
3466         Q ( ( P '\\\\' + 1 - S "\r" ) ^ 1 )
3467         +
3468         EOL
3469       ) ^ 0
3470       * Q "\""
3471     )

```

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 programming, see the paragraphe *Lua’s long strings* in www.inf.puc-rio.br/~roberto/lpeg.

```

3472 local ext = ( R "az" + "_" ) ^ 0
3473 local open = "{" * Cg ( ext , 'init' ) * "/"
3474 local close = "/" * C ( ext ) * "}"
3475 local closeeq =
3476   Cmt ( close * Cb ( 'init' ) ,
3477     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3478 local QuotedStringBis =
3479   WithStyle ( 'String.Long.Internal' ,
3480     (
3481       Space
3482       +
3483       Q ( ( 1 - S "\r" ) ^ 1 )
3484       +
3485       EOL
3486     ) ^ 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.

```

3487 local QuotedString =
3488   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3489   ( 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).

```

3490 local comment =
3491   P {
3492     "A" ,
3493     A = Q "(*"
3494       * ( V "A"
3495         + Q ( ( 1 - S "\r$" - "(*" - "*)" ) ^ 1 ) -- $
3496         + Q ( ocaml_string )
3497         + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3498         + EOL
3499       ) ^ 0
3500       * Q "*)"
3501   }
3502 local Comment = WithStyle ( 'Comment.Internal' , comment )

```

Some standard LPEG

```
3503 local Delim = K ( 'Delim' , P "[" + "]" + S "()" )
3504 local Punct = K ( 'Punct' , 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.

```
3505 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.

```
3506 local Constructor =
3507   P "::"
```

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

```
3508 * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3509 +
3510 P "[]"
3511 * Lc ([{\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]])
3512 K ( 'Name.Constructor' ,
3513     Q "`" ^ -1 * cap_identifier
3514     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )
```

```
3515 local ModuleType = K ( 'Name.Type' , cap_identifier )
```

```
3516 local OperatorWord =
3517   K ( 'Operator.Word' ,
3518       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )
```

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

```
3519 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3520   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3521   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3522   "struct" + "type" + "val"
```

```
3523 local Keyword =
3524   K ( 'Keyword' ,
3525       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3526       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3527       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3528       + "virtual" + "when" + "while" + "with" )
3529   + K ( 'Keyword.Constant' , P "true" + "false" )
3530   + K ( 'Keyword.Governing' , governing_keyword )
```

```
3531 local EndKeyword
3532   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3533   + EscapeMath + -1
```

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

```
3534 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3535   - ( 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`.

```
3536 local Identifier = K ( 'Identifier.Internal' , identifier )
```

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

```

3537 local ocaml_char =
3538   P "'" *
3539   (
3540     ( 1 - S "\\\" )
3541     + "\\\"
3542     * ( S "\\ntbr \"\"
3543         + digit * digit * digit
3544         + P "x" * ( digit + R "af" + R "AF" )
3545                   * ( digit + R "af" + R "AF" )
3546                   * ( digit + R "af" + R "AF" )
3547         + P "o" * R "03" * R "07" * R "07" )
3548   )
3549   * "'"
3550 local Char =
3551   K ( 'String.Short.Internal', ocaml_char )

```

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

```

3552 local TypeParameter =
3553   K ( 'TypeParameter' ,
3554       "'" * 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.

```

3555 local DotNotation =
3556   (
3557     K ( 'Name.Module' , cap_identifier )
3558       * Q "."
3559       * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3560       +
3561       Identifier
3562       * Q "."
3563       * K ( 'Name.Field' , identifier )
3564   )
3565   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

```

The records

```

3566 local expression_for_fields_type =
3567   P { "E" ,
3568       E = ( "{ " * V "F" * "}"
3569           + "(" * V "F" * ")"
3570           + TypeParameter
3571           + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
3572       F = ( "{ " * V "F" * "}"
3573           + "(" * V "F" * ")"
3574           + ( 1 - S "{}()[]\r\" ) + TypeParameter ) ^ 0
3575   }

```

```

3576 local expression_for_fields_value =
3577   P { "E" ,
3578       E = ( "{ " * V "F" * "}"
3579           + "(" * V "F" * ")"
3580           + "[" * V "F" * "]"
3581           + ocaml_string + ocaml_char
3582           + ( 1 - S "{}()[];" ) ) ^ 0 ,
3583       F = ( "{ " * V "F" * "}"
3584           + "(" * V "F" * ")"
3585           + "[" * V "F" * "]"
3586           + ocaml_string + ocaml_char
3587           + ( 1 - S "{}()[]\" ) ) ^ 0
3588   }

```

```

3589 local OneFieldDefinition =
3590   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3591   * K ( 'Name.Field' , identifier ) * SkipSpace
3592   * Q ":" * SkipSpace
3593   * K ( 'TypeExpression' , expression_for_fields_type )
3594   * SkipSpace

```

```

3595 local OneField =
3596   K ( 'Name.Field' , identifier ) * SkipSpace
3597   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3598   * ( C ( expression_for_fields_value ) / ParseAgain )
3599   * SkipSpace

```

The records.

```

3600 local RecordVal =
3601   Q "{" * SkipSpace
3602   *
3603   (
3604     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3605   ) ^ -1
3606   *
3607   (
3608     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3609   )
3610   * SkipSpace
3611   * Q ";" ^ -1
3612   * SkipSpace
3613   * Comment ^ -1
3614   * SkipSpace
3615   * Q "}"
3616 local RecordType =
3617   Q "{" * SkipSpace
3618   *
3619   (
3620     OneFieldDefinition
3621     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3622   )
3623   * SkipSpace
3624   * Q ";" ^ -1
3625   * SkipSpace
3626   * Comment ^ -1
3627   * SkipSpace
3628   * Q "}"
3629 local Record = RecordType + RecordVal

```

```

3630 local Operator =
3631   P "||" *

```

Don't use \hspace instead of \kern!

```

3632 Lc([{\PitonStyle{Operator}{\kern0.1em/\kern-0.2em/\kern0.1em}}])
3633 +
3634 K ( 'Operator' ,
3635   P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3636   "//" + "*" + ";" + "->" + "+." + "-." + "*." + "/" +
3637   + S "--+/*%=<>&@|" )

```

```

3638 local Builtin =
3639   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

```



```

3640 local Exception =
3641   K ( 'Exception' ,
3642     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3643     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3644     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

3645 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).

```

3646 local pattern_part =
3647   ( 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).

```
3648 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.

```

3649   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3650   *

```

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

```

3651   (
3652     K ( 'Identifier.Internal' , identifier )
3653     +
3654     Q "(" * SkipSpace
3655     * ( C ( pattern_part ) / ParseAgain )
3656     * SkipSpace

```

Of course, the specification of type is optional.

```

3657     * ( Q ":" * #(1- P"=")
3658         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3659     ) ^ -1
3660     * Q ")"
3661   )

```

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

```

3662 local DefFunction =
3663   K ( 'Keyword.Governing' , "let open" )
3664   * Space
3665   * K ( 'Name.Module' , cap_identifier )
3666   +
3667   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3668   * Space
3669   * K ( 'Name.Function.Internal' , identifier )
3670   * Space
3671   * (

```

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).

```

3672   Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3673   +
3674   Argument * ( SkipSpace * Argument ) ^ 0
3675   * (
3676     SkipSpace
3677     * Q ":" * #( 1 - P "=" )
3678     * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3679   ) ^ -1
3680   )

```

DefModule

```

3681 local DefModule =
3682   K ( 'Keyword.Governing' , "module" ) * Space
3683   *
3684   (
3685     K ( 'Keyword.Governing' , "type" ) * Space
3686     * K ( 'Name.Type' , cap_identifier )
3687     +
3688     K ( 'Name.Module' , cap_identifier ) * SkipSpace
3689     *
3690     (
3691       Q "(" * SkipSpace
3692       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3693       * Q ":" * # ( 1 - P "=" ) * SkipSpace
3694       * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3695       *
3696       (
3697         Q "," * SkipSpace
3698         * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3699         * Q ":" * # ( 1 - P "=" ) * SkipSpace
3700         * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3701       ) ^ 0
3702       * Q ")"
3703     ) ^ -1
3704   *
3705   (
3706     Q "=" * SkipSpace
3707     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3708     * Q "("
3709     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3710     *
3711     (
3712       Q ","
3713       *
3714       K ( 'Name.Module' , cap_identifier ) * SkipSpace
3715     ) ^ 0
3716     * Q ")"
3717   ) ^ -1
3718 )
3719 +
3720 K ( 'Keyword.Governing' , P "include" + "open" )
3721 * Space
3722 * K ( 'Name.Module' , cap_identifier )

```

DefType

```

3723 local DefType =
3724   K ( 'Keyword.Governing' , "type" )
3725   * Space
3726   * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3727   * SkipSpace
3728   * ( Q "+=" + Q "=" )
3729   * SkipSpace
3730   * (
3731     RecordType
3732     +

```

The following lines are a suggestion of Y. Salmon.

```

3733 WithStyle
3734 (
3735   'TypeExpression' ,
3736   (
3737     (
3738       EOL

```

```

3739         + comment
3740         + Q ( 1
3741             - P ";;"
3742             - P "type"
3743             - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3744         )
3745     ) ^ 0
3746     *
3747     (
3748         # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3749         + Q ";;"
3750         + -1
3751     )
3752 )
3753 )
3754 )

3755 local prompt =
3756   Q "utop[" * digit^1 * Q "]"> "
3757 local start_of_line = P(function(subject, position)
3758   if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3759     return position
3760   end
3761   return nil
3762 end)
3763 local Prompt = #start_of_line * K( 'Prompt', prompt )
3764 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3765               * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3766               * ( K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

The main LPEG for the language OCaml

```

3767 local Main =
3768   space ^ 0 * EOL
3769   + Space
3770   + Tab
3771   + Escape + EscapeMath
3772   + Beamer
3773   + DetectedCommands
3774   + TypeParameter
3775   + String + QuotedString + Char
3776   + Comment
3777   + Prompt + Answer

```

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

```

3778   + Q "~" * Identifier * ( Q ":" ) ^ -1
3779   + Q ":" * # (1 - P ":" ) * SkipSpace
3780       * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3781   + Exception
3782   + DefType
3783   + DefFunction
3784   + DefModule
3785   + Record
3786   + Keyword * EndKeyword
3787   + OperatorWord * EndKeyword
3788   + Builtin * EndKeyword
3789   + DotNotation * EndKeyword
3790   + Constructor
3791   + Identifier
3792   + Punct
3793   + Delim -- Delim is before Operator for a correct analysis of [| et |]
3794   + Operator

```

```

3795     + Number
3796     + Word

```

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

```

3797     LPEG1.ocaml = Main ^ 0

```

```

3798     LPEG2.ocaml =
3799     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).

```

3800         (
3801         (
3802             P ":"
3803             +
3804             (
3805                 ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3806                 * Identifier
3807                 * SkipSpace
3808                 * Q ":"
3809             )
3810         )
3811         * # ( 1 - S ":@" )
3812         * SkipSpace
3813         * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 ) * -1
3814     )
3815     +
3816     (
3817         ( space ^ 0 * "\r" ) ^ -1
3818         * Lc [[ \@@_begin_line: ]]
3819         * LeadingSpace ^ 0
3820         * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
3821           + space ^ 0 * EOL
3822           + Main
3823         ) ^ 0
3824         * -1
3825         * Lc [[ \@@_end_line: ]]
3826     )
3827 )

```

End of the Lua scope for the language OCaml.

```

3828 end

```

3.6 The language C

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

```

3829 --c C c++ C++
3830 do

3831     local Delim = Q ( S "{[()]} " )
3832     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.

```

3833     local identifier = letter * alphanum ^ 0
3834
3835     local Operator =

```

```

3836     K ( 'Operator' ,
3837         P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3838         + S "~+/*%=<>&.@|!" )
3839
3840     local Keyword =
3841         K ( 'Keyword' ,
3842             P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3843             "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3844             "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3845             "register" + "restricted" + "return" + "static" + "static_assert" +
3846             "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3847             "union" + "using" + "virtual" + "volatile" + "while"
3848         )
3849         + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3850
3851     local Builtin =
3852         K ( 'Name.Builtin' ,
3853             P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3854
3855     local Type =
3856         K ( 'Name.Type' ,
3857             P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3858             "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3859             "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3860             "void" + "wchar_t" ) * Q "*" ^ 0
3861
3862     local DefFunction =
3863         Type
3864         * Space
3865         * Q "*" ^ -1
3866         * K ( 'Name.Function.Internal' , identifier )
3867         * SkipSpace
3868         * # 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:`

```

3869     local DefClass =
3870         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).

```

3871     local Character =
3872         K ( 'String.Short' ,
3873             P [['\']] + P "'" * ( 1 - P "'" ) ^ 0 * P "'" )

```

The strings of C

```

3874     String =
3875         WithStyle ( 'String.Long.Internal' ,
3876             Q "\""
3877             * ( SpaceInString
3878                 + K ( 'String.Interpol' ,
3879                     "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3880                 )
3881             + Q ( ( P "\\\"" + 1 - S "\"" ) ^ 1 )
3882             ) ^ 0
3883             * Q "\""
3884         )

```

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

```

3885 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
3886 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end

3887 DetectedCommands =
3888   Compute_DetectedCommands ( 'c' , braces )
3889   + Compute_RawDetectedCommands ( 'c' , braces )

3890 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

The directives of the preprocessor

```

3891 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.

```

3892 local Comment =
3893   WithStyle ( 'Comment.Internal' ,
3894     Q "/" * ( CommentMath + Q ( ( 1 - S "$r" ) ^ 1 ) ) ^ 0 ) -- $
3895     * ( EOL + -1 )
3896
3897 local LongComment =
3898   WithStyle ( 'Comment.Internal' ,
3899     Q "/*"
3900     * ( CommentMath + Q ( ( 1 - P "*/" - S "$r" ) ^ 1 ) + EOL ) ^ 0
3901     * Q "*/"
3902     ) -- $

```

The main LPEG for the language C

```

3903 local EndKeyword
3904   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3905     EscapeMath + -1

```

First, the main loop :

```

3906 local Main =
3907   space ^ 0 * EOL
3908   + Space
3909   + Tab
3910   + Escape + EscapeMath
3911   + CommentLaTeX
3912   + Beamer
3913   + DetectedCommands
3914   + Preproc
3915   + Comment + LongComment
3916   + Delim
3917   + Operator
3918   + Character
3919   + String
3920   + Punct
3921   + DefFunction
3922   + DefClass
3923   + Type * ( Q "*" ^ -1 + EndKeyword )
3924   + Keyword * EndKeyword
3925   + Builtin * EndKeyword
3926   + Identifier
3927   + Number
3928   + Word

```

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

```
3929 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.

```
3930 LPEG2.c =
3931 Ct (
3932   ( space ^ 0 * P "\r" ) ^ -1
3933   * Lc [[ \@@_begin_line: ]]
3934   * LeadingSpace ^ 0
3935   * ( space ^ 1 * -1 + Main ) ^ 0
3936   * -1
3937   * Lc [[ \@@_end_line: ]]
3938 )
```

End of the Lua scope for the language C.

```
3939 end
```

3.7 The language SQL

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

```
3940 --sql SQL
3941 do
3942   local LuaKeyword
3943   function LuaKeyword ( name ) return
3944     Lc [[ {\PitonStyle{Keyword}{ }}
3945     * Q ( Cmt (
3946       C ( letter * alphanum ^ 0 ) ,
3947       function ( _ , _ , a ) return a : upper ( ) == name end
3948     )
3949   )
3950   * Lc "}"
3951 end
```

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

```
3952 local identifier =
3953   letter * ( alphanum + "-" ) ^ 0
3954   + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"
3955 local Operator =
3956   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).

```
3957 local Set
3958 function Set ( list )
3959   local set = { }
3960   for _ , l in ipairs ( list ) do set[l] = true end
3961   return set
3962 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.

```

3963 local set_keywords = Set
3964 {
3965     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3966     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3967     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3968     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3969     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3970     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3971     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3972     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3973     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3974     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3975     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3976     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
3977     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3978     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3979     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3980     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3981     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3982     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3983     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3984     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3985 }
3986
3987 local set_builtins = Set
3988 {
3989     "AVG", "COUNT", "CHAR_LENGTH", "CONCAT", "CURDATE", "CURRENT_DATE",
3990     "DATE_FORMAT", "DAY", "LOWER", "LTRIM", "MAX", "MIN", "MONTH", "NOW",
3991     "RANK", "ROUND", "RTRIM", "SUBSTRING", "SUM", "UPPER", "YEAR"
3992 }

```

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.

```

3992 local Identifier =
3993   C ( identifier ) /
3994   (
3995     function ( s )
3996       if set_keywords [ s : upper ( ) ] then return

```

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

```

3997     { [[{\PitonStyle{Keyword}{}}] } ,
3998     { luatexbase.catcodetables.other , s } ,
3999     { "}" }
4000   else
4001     if set_builtins [ s : upper ( ) ] then return
4002     { [[{\PitonStyle{Name.Builtin}{}}] } ,
4003     { luatexbase.catcodetables.other , s } ,
4004     { "}" }
4005   else return
4006   { [[{\PitonStyle{Name.Field}{}}] } ,
4007   { luatexbase.catcodetables.other , s } ,
4008   { "}" }
4009   end
4010 end
4011 end
4012 )

```

The strings of SQL

```

4013 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.

```

4014 local braces = Compute_braces ( "'" * ( 1 - P "'" ) ^ 1 * "'" )
4015 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end

4016 DetectedCommands =
4017   Compute_DetectedCommands ( 'sql' , braces )
4018   + Compute_RawDetectedCommands ( 'sql' , braces )
4019 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.

```

4020 local Comment =
4021   WithStyle ( 'Comment.Internal' ,
4022     Q "--" -- syntax of SQL92
4023     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
4024     * ( EOL + -1 )
4025
4026 local LongComment =
4027   WithStyle ( 'Comment.Internal' ,
4028     Q "/*"
4029     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
4030     * Q "*/"
4031     ) -- $

```

The main LPEG for the language SQL

```

4032 local EndKeyword
4033   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
4034     EscapeMath + -1
4035
4036 local TableField =
4037   K ( 'Name.Table' , identifier )
4038   * Q "."
4039   * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
4040
4041 local OneField =
4042   (
4043     Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
4044     +
4045     K ( 'Name.Table' , identifier )
4046     * Q "."
4047     * K ( 'Name.Field' , identifier )
4048     +
4049     K ( 'Name.Field' , identifier )
4050   )
4051   * (
4052     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
4053     ) ^ -1
4054   * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
4055
4056 local OneTable =
4057   K ( 'Name.Table' , identifier )
4058   * (
4059     Space
4060     * LuaKeyword "AS"
4061     * Space
4062     * K ( 'Name.Table' , identifier )
4063     ) ^ -1
4064
4065 local WeCatchTableNames =

```

```

4065     LuaKeyword "FROM"
4066     * ( Space + EOL )
4067     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
4068     + (
4069         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
4070         + LuaKeyword "TABLE"
4071     )
4072     * ( Space + EOL ) * OneTable
4073     local EndKeyword
4074     = Space + Punct + Delim + EOL + Beamer
4075     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

4076     local Main =
4077         space ^ 0 * EOL
4078         + Space
4079         + Tab
4080         + Escape + EscapeMath
4081         + CommentLaTeX
4082         + Beamer
4083         + DetectedCommands
4084         + Comment + LongComment
4085         + Delim
4086         + Operator
4087         + String
4088         + Punct
4089         + WeCatchTableNames
4090         + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
4091         + Number
4092         + Word

```

Here, we must not put local, of course.

```

4093     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.

```

4094     LPEG2.sql =
4095         Ct (
4096             ( space ^ 0 * "\r" ) ^ -1
4097             * Lc [[ \@@_begin_line: ]]
4098             * LeadingSpace ^ 0
4099             * ( space ^ 1 * -1 + Main ) ^ 0
4100             * -1
4101             * Lc [[ \@@_end_line: ]]
4102         )

```

End of the Lua scope for the language SQL.

```

4103 end

```

3.8 The language “Minimal”

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

```

4104 --minimal Minimal
4105 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:`

```

4106 local Punct = Q ( S ",:;!\\\" )
4107
4108 local Comment =
4109   WithStyle ( 'Comment.Internal' ,
4110     Q "\""
4111     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4112     )
4113     * ( EOL + -1 )
4114
4115 local String =
4116   WithStyle ( 'String.Short.Internal' ,
4117     Q "\""
4118     * ( SpaceInString
4119       + Q ( ( P "[\]" ) + 1 - S " \"\" ) ^ 1 )
4120     ) ^ 0
4121     * Q "\""
4122   )

```

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

```

4123 local braces = Compute_braces ( P "\"" * ( P "\\\"" + 1 - P "\"" ) ^ 1 * "\"" )
4124
4125 if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
4126
4127 DetectedCommands =
4128   Compute_DetectedCommands ( 'minimal' , braces )
4129   + Compute_RawDetectedCommands ( 'minimal' , braces )
4130
4131 LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
4132
4133 local identifier = letter * alphanum ^ 0
4134
4135 local Identifier = K ( 'Identifier.Internal' , identifier )
4136
4137 local Delim = Q ( S "{[()]}" )
4138
4139 local Main =
4140   space ^ 0 * EOL
4141   + Space
4142   + Tab
4143   + Escape + EscapeMath
4144   + CommentLaTeX
4145   + Beamer
4146   + DetectedCommands
4147   + Comment
4148   + Delim
4149   + String
4150   + Punct
4151   + Identifier
4152   + Number
4153   + Word

```

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

```

4154 LPEG1.minimal = Main ^ 0
4155
4156 LPEG2.minimal =
4157   Ct (
4158     ( space ^ 0 * "\r" ) ^ -1
4159     * Lc [ [ @@_begin_line: ] ]
4160     * LeadingSpace ^ 0
4161     * ( space ^ 1 * -1 + Main ) ^ 0
4162     * -1

```

```

4163         * Lc [[ \@@_end_line: ]]
4164     )

```

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

```

4165 end

```

3.9 The language “Verbatim”

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

```

4166 --verbatim Verbatim
4167 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”).

```

4168     local braces =
4169         P { "E" ,
4170             E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4171         }
4172
4173     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
4174
4175     DetectedCommands =
4176         Compute_DetectedCommands ( 'verbatim' , braces )
4177         + Compute_RawDetectedCommands ( 'verbatim' , braces )
4178
4179     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

4180     local lpeg_central = 1 - S " \\r"
4181     if piton.begin_escape then
4182         lpeg_central = lpeg_central - piton.begin_escape
4183     end
4184     if piton.begin_escape_math then
4185         lpeg_central = lpeg_central - piton.begin_escape_math
4186     end
4187     local Word = Q ( lpeg_central ^ 1 )
4188
4189     local Main =
4190         space ^ 0 * EOL
4191         + Space
4192         + Tab
4193         + Escape + EscapeMath
4194         + Beamer
4195         + DetectedCommands
4196         + Q [[\]]
4197         + Word

```

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

```

4198     LPEG1.verbatim = Main ^ 0
4199
4200     LPEG2.verbatim =
4201         Ct (
4202             ( space ^ 0 * "\r" ) ^ -1
4203             * Lc [[ \@@_begin_line: ]]
4204             * LeadingSpace ^ 0
4205             * ( space ^ 1 * -1 + Main ) ^ 0
4206             * -1
4207             * Lc [[ \@@_end_line: ]]
4208         )

```

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

```

4209 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).

```

4210 --EXPL expl
4211 do
4212     local Comment =
4213         WithStyle
4214         ( 'Comment.Internal' ,
4215         Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4216         )
4217         * ( EOL + -1 )

```

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

```

4218     local analyze_cs
4219     function analyze_cs ( s )
4220         local i = s : find ( ":" )
4221         if i then

```

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

```

4222         local name = s : sub ( 2 , i - 1 )
4223         local parts = name : explode ( "_" )
4224         local module = parts[1]
4225         if module == "" then module = parts[3] end

```

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

```

4226         return
4227         { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
4228         { luatexbase.catcodetables.other , s } ,
4229         { "}" } }
4230     else
4231         local p = s : sub ( 1 , 3 )
4232         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`.

```

4233         local scope = s : sub(2,2)
4234         local parts = s : explode ( "_" )
4235         local module = parts[2]
4236         if module == "" then module = parts[3] end
4237         local type = parts[#parts]
4238         return
4239         { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
4240         { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
4241         { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,
4242         { luatexbase.catcodetables.other , s } ,
4243         { "}}}}}} }
4244     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.

```

4245         return { luatexbase.catcodetables.other , s }
4246     end
4247 end
4248 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`).

```

4249     local braces =
4250     P { "E" ,
4251     E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4252     }

```

```

4253
4254 if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
4255
4256 DetectedCommands =
4257   Compute_DetectedCommands ( 'expl' , braces )
4258   + Compute_RawDetectedCommands ( 'expl' , braces )
4259
4260 LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
4261 local control_sequence = P "\\\" * ( R "Az" + "_" + ":" + "@" ) ^ 1
4262 local ControlSequence = C ( control_sequence ) / analyze_cs
4263
4264 local def_function
4265   = P [[\cs_]]
4266   * ( P "set" + "new" )
4267   * ( P "_protected" ) ^ -1
4268   * P ":N" * ( P "p" ) ^ -1 * "n"
4269
4270 local DefFunction =
4271   C ( def_function ) / analyze_cs
4272   * Space
4273   * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
4274   * ControlSequence -- Q ( ControlSequence ) ?
4275   * Lc "}"
4276
4277 local Word = Q ( ( 1 - S " \r" ) ^ 1 )
4278
4279 local Main =
4280   space ^ 0 * EOL
4281   + Space
4282   + Tab
4283   + Escape + EscapeMath
4284   + Beamer
4285   + Comment
4286   + DetectedCommands
4287   + DefFunction
4288   + ControlSequence
4289   + Word

```

Here, we must not put local, of course.

```

4287 LPEG1.expl = Main ^ 0
4288
4289 LPEG2.expl =
4290   Ct (
4291     ( space ^ 0 * "\r" ) ^ -1
4292     * Lc [[ \@@_begin_line: ] ]
4293     * LeadingSpace ^ 0
4294     * ( space ^ 1 * -1 + Main ) ^ 0
4295     * -1
4296     * Lc [[ \@@_end_line: ] ]
4297   )

```

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

```

4298 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.

```

4299 function piton.Parse ( language , code )

```

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

```

4300 piton.language = language
4301 local t = LPEG2[language] : match ( code )
4302 if not t then
4303     sprintL3 ([" @@_error_or_warning:n { SyntaxError } ])
4304     return -- to exit in force the function
4305 end
4306 local left_stack = {}
4307 local right_stack = {}
4308 for _ , one_item in ipairs ( t ) do
4309     if one_item == "EOL" then
4310         for i = #right_stack, 1, -1 do
4311             tex.sprint ( right_stack[i] )
4312         end

```

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

```

4313     sprintL3 ( [" @@_end_line: @@_par: @@_begin_line: ] ] )
4314     tex.sprint ( table.concat ( left_stack ) )
4315 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}`.

```

4316     if one_item[1] == "Open" then
4317         tex.sprint ( one_item[2] )
4318         table.insert ( left_stack , one_item[2] )
4319         table.insert ( right_stack , one_item[3] )
4320     else
4321         if one_item[1] == "Close" then
4322             tex.sprint ( right_stack[#right_stack] )
4323             left_stack[#left_stack] = nil
4324             right_stack[#right_stack] = nil
4325         else
4326             tex.tprint ( one_item )
4327         end
4328     end
4329 end
4330 end
4331 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).

```

4332 local my_file_lines
4333 function my_file_lines ( filename )
4334     local f = io.open ( filename , 'rb' )
4335     local s = f : read ( '*a' )
4336     f : close ( )

```

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

```

4337     return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
4338 end

```

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

```

4339 function piton.ReadFile ( name , first_line , last_line )
4340     local s = ''
4341     local i = 0
4342     for line in my_file_lines ( name ) do

```

```

4343     i = i + 1
4344     if i >= first_line then
4345         s = s .. '\r' .. line
4346     end
4347     if i >= last_line then break end
4348 end

```

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

```

4349     if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
4350         s = s : sub ( 5 , -1 )
4351     end

4352     sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { }}] )
4353     tex.sprint ( luatexbase.catcodetables.other , s )
4354     sprintL3 ( "}" )
4355 end

4356 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
4357     local s
4358     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4359     piton.GobbleParse ( lang , n , splittable , s )
4360 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 `#`.

```

4361 function piton.ParseBis ( lang , code )
4362     return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
4363 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.

```

4364 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:`.

```

4365     return piton.Parse
4366     (
4367         lang ,
4368         code : gsub ( [[\@@_breakable_space: ]] , ' ' )
4369     )
4370 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.

```

4371 local AutoGobbleLPEG =
4372     ( (
4373         P " " ^ 0 * "\r"

```



```

4374      +
4375      Ct ( C " " ^ 0 ) / table.getn
4376      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
4377    ) ^ 0
4378    * ( Ct ( C " " ^ 0 ) / table.getn
4379      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4380  ) / math.min

```

The following LPEG is similar but works with the tabulations.

```

4381 local TabsAutoGobbleLPEG =
4382 (
4383   (
4384     P "\t" ^ 0 * "\r"
4385     +
4386     Ct ( C "\t" ^ 0 ) / table.getn
4387     * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
4388   ) ^ 0
4389   * ( Ct ( C "\t" ^ 0 ) / table.getn
4390     * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4391 ) / 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).

```

4392 local EnvGobbleLPEG =
4393   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
4394   * 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.

```

4395 function piton.Gobble ( n , code )
4396   if n == 0 then return
4397     code
4398   else
4399     if n == -1 then
4400       n = AutoGobbleLPEG : match ( code )

```

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

```

4401     if tonumber(n) then else n = 0 end
4402   else
4403     if n == -2 then
4404       n = EnvGobbleLPEG : match ( code )
4405     else
4406       if n == -3 then
4407         n = TabsAutoGobbleLPEG : match ( code )
4408         if tonumber(n) then else n = 0 end
4409       end
4410     end
4411   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...

```

4412     if n == 0 then return
4413       code
4414     else return

```

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

```

4415   ( Ct (
4416     ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4417     * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4418   ) ^ 0 )
4419   / table.concat

```

```

4420         ) : match ( code )
4421     end
4422 end
4423 end

```

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

```

4424 function piton.GobbleParse ( lang , n , splittable , code )
4425     piton.ComputeLinesStatus ( code , splittable )
4426     piton.last_code = piton.Gobble ( n , code )
4427     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`.

```

4428     piton.CountLines ( piton.last_code )
4429     piton.Parse ( lang , piton.last_code )
4430     piton.join_and_write ( )
4431 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`.

```

4432 function piton.join_and_write ( )
4433     if piton.join ~= '' then
4434         if not piton.join_files [ piton.join ] then
4435             piton.join_files [ piton.join ] = piton.get_last_code ( )
4436         else
4437             if piton.join_separation == '' then
4438                 piton.join_files [ piton.join ] =
4439                     piton.join_files [ piton.join ]
4440                     .. "\r\n"
4441                 .. piton.get_last_code ( )
4442             else
4443                 piton.join_files [ piton.join ] =
4444                     piton.join_files [ piton.join ]
4445                     .. "\r\n"
4446                     .. ( piton.join_separation : gsub ( '##' , '#' ) )
4447                     .. "\r\n"
4448                     .. piton.get_last_code ( )
4449             end
4450         end
4451     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.

```

4452     if piton.write ~= '' then

```

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

```

4453         local file_name = ''
4454         if piton.path_write == '' then
4455             file_name = piton.write
4456         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.

```

4457             local attr = lfs.attributes ( piton.path_write )
4458             if attr and attr.mode == "directory" then
4459                 file_name = piton.path_write .. "/" .. piton.write
4460             else

```

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

```

4461      sprintL3 [[ \@_error_or_warning:n { InexistentDirectory } ]]
4462    end
4463  end
4464  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`).

```

4465      if not piton.write_files [ file_name ] then
4466        piton.write_files [ file_name ] = piton.get_last_code ( )
4467      else
4468        piton.write_files [ file_name ] =
4469          piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4470      end
4471    end
4472  end
4473 end

```

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

```

4474 function piton.GobbleParseNoPrint ( lang , n , code )
4475   piton.last_code = piton.Gobble ( n , code )
4476   piton.last_language = lang
4477   piton.join_and_write ( )
4478 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).

```

4479 function piton.GobbleSplitParse ( lang , n , splittable , code )
4480   local chunks
4481   chunks =
4482     (
4483       Ct (
4484         (
4485           P " " ^ 0 * "\r"
4486         +
4487           C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4488             - ( P " " ^ 0 * ( P "\r" + -1 ) )
4489           ) ^ 1
4490         )
4491       ) ^ 0
4492     )
4493   ) : match ( piton.Gobble ( n , code ) )
4494   sprintL3 [[ \begingroup ]]
4495   sprintL3
4496     (
4497       [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
4498       .. "language = " .. lang .. ","
4499       .. "splittable = " .. splittable .. "}"]
4500     )
4501   for k , v in pairs ( chunks ) do
4502     if k > 1 then
4503       sprintL3 ( [[ \l_@_split_separation_tl ]] )
4504     end
4505     tex.print
4506       (
4507         [[\begin{}} .. piton.env_used_by_split .. "}\r"

```

```

4508         .. v
4509         .. [[\end{}} .. piton.env_used_by_split .. "}\r"
4510     )
4511 end
4512 sprintL3 [[ \endgroup ]]
4513 end

4514 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4515     local s
4516     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4517     piton.GobbleSplitParse ( lang , n , splittable , s )
4518 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.

```

4519 piton.string_between_chunks =
4520 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4521 .. [[ \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.

```

4522 function piton.get_last_code ( )
4523     return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4524     : gsub ( '\r\n?' , '\n' )
4525 end

```

3.14 To count the number of lines

```

4526 local CountBeamerEnvironments
4527 function CountBeamerEnvironments ( code ) return
4528 (
4529     Ct (
4530         (
4531             P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4532             +
4533             ( 1 - P "\r" ) ^ 0 * "\r"
4534         ) ^ 0
4535         * ( 1 - P "\r" ) ^ 0
4536         * -1
4537     ) / table.getn
4538 ) : match ( code )
4539 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.

```

4540 function piton.CountLines ( code )
4541     local count
4542     count =
4543         ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4544             *
4545             (
4546                 space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4547                 + space ^ 0
4548             ) ^ -1
4549             * -1

```

```

4550         ) / table.getn
4551     ) : match ( code )
4552     if piton.beamer then
4553         count = count - 2 * CountBeamerEnvironments ( code )
4554     end
4555     sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4556 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).

```

4557 function piton.CountNonEmptyLines ( code )
4558     local count = 0

```

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

```

4559     count =
4560         ( Ct ( ( P " " ^ 0 * "\r"
4561             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4562             * ( 1 - P "\r" ) ^ 0
4563             * -1
4564             ) / table.getn
4565         ) : match ( code )
4566     count = count + 1
4567     if piton.beamer then
4568         count = count - 2 * CountBeamerEnvironments ( code )
4569     end
4570     sprintL3
4571     ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4572 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.

```

4573 function piton.ComputeRange ( s , t , file_name )
4574     local first_line = -1
4575     local count = 0
4576     local last_found = false
4577     for line in io.lines ( file_name ) do
4578         if first_line == -1 then
4579             if line : sub ( 1 , #s ) == s then
4580                 first_line = count
4581             end
4582         else
4583             if line : sub ( 1 , #t ) == t then
4584                 last_found = true
4585                 break
4586             end
4587         end
4588         count = count + 1
4589     end
4590     if first_line == -1 then
4591         sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4592     else
4593         if not last_found then
4594             sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4595         end
4596     end
4597     sprintL3 (
4598         [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 '
4599         .. [[ \global \l_@@_last_line_int = ]] .. count )
4600 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`.

```
4601 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).

```
4602     local lpeg_line_beamer
4603     if piton.beamer then
4604         lpeg_line_beamer =
4605             space ^ 0
4606             * P [[\begin{]] * beamerEnvironments * "]"
4607             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4608             +
4609             space ^ 0
4610             * P [[\end{]] * beamerEnvironments * "]"
4611     else
4612         lpeg_line_beamer = P ( false )
4613     end
4614     local lpeg_empty_lines =
4615         Ct (
4616             ( lpeg_line_beamer * "\r"
4617             +
4618             P " " ^ 0 * "\r" * Cc ( 0 )
4619             +
4620             ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4621             ) ^ 0
4622             *
4623             ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4624         )
4625         * -1
4626     local lpeg_all_lines =
4627         Ct (
4628             ( lpeg_line_beamer * "\r"
4629             +
4630             ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4631             ) ^ 0
4632             *
4633             ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4634         )
4635         * -1
```

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

```
4636     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`).

```

4637 local lines_status
4638 local s = splittable
4639 if splittable < 0 then s = - splittable end

4640 if splittable > 0 then
4641   lines_status = lpeg_all_lines : match ( code )
4642 else

```

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

```

4643   lines_status = lpeg_empty_lines : match ( code )
4644   for i , x in ipairs ( lines_status ) do
4645     if x == 0 then
4646       for j = 1 , s - 1 do
4647         if i + j > #lines_status then break end
4648         if lines_status[i+j] == 0 then break end
4649         lines_status[i+j] = 2
4650       end
4651       for j = 1 , s - 1 do
4652         if i - j == 1 then break end
4653         if lines_status[i-j-1] == 0 then break end
4654         lines_status[i-j-1] = 2
4655       end
4656     end
4657   end
4658 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.

```

4659   for j = 1 , s - 1 do
4660     if j > #lines_status then break end
4661     if lines_status[j] == 0 then break end
4662     lines_status[j] = 2
4663   end

```

Now, from the end of the code.

```

4664   for j = 1 , s - 1 do
4665     if #lines_status - j == 0 then break end
4666     if lines_status[#lines_status - j] == 0 then break end
4667     lines_status[#lines_status - j] = 2
4668   end

```

```

4669   piton.lines_status = lines_status
4670 end

```

```

4671 function piton.TranslateBeamerEnv ( code )
4672   local s
4673   s =
4674   (
4675     Ct (
4676       (
4677         space ^ 0
4678         * C (
4679           ( P "\\begin{" + "\\end{" )
4680           * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * "\r"
4681         )
4682         + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4683       ) ^ 0
4684     *
4685     (

```

```

4686      (
4687          space ^ 0
4688          * C (
4689              ( P "\\begin{" + "\\end{" )
4690              * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * -1
4691          )
4692          + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4693      ) ^ -1
4694  )
4695  ) ^ -1 / table.concat
4696  ) : match ( code )
4697  sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4698  tex.sprint ( luatexbase.catcodetables.other , s )
4699  sprintL3 ( "]" )
4700 end

```

3.16 To create new languages with the syntax of listings

```

4701 function piton.new_language ( lang , definition )
4702     lang = lang : lower ( )

4703     local alpha , digit = lpeg.alpha , lpeg.digit
4704     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`.

```

4705     function add_to_letter ( c )
4706         if c ~= " " then table.insert ( extra_letters , c ) end
4707     end

```

For the digits, it's straitforward.

```

4708     function add_to_digit ( c )
4709         if c ~= " " then digit = digit + c end
4710     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 `}`).

```

4711     local other = S "._@+~*/<>!?.() [] ~^=#&\"'\\$" --
4712     local extra_others = { }
4713     function add_to_other ( c )
4714         if c ~= " " then

```

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

```

4715         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 `</...>`.

```

4716         other = other + P ( c )
4717     end
4718 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`.

```

4719     local def_table
4720     if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4721         def_table = {}
4722     else
4723         local strict_braces =

```



```

4724     P { "E" ,
4725         E = ( "{" * V "F" * "}" + ( 1 - S ",{" }" ) ) ^ 0 ,
4726         F = ( "{" * V "F" * "}" + ( 1 - S "{" }" ) ) ^ 0
4727     }
4728     local cut_definition =
4729     P { "E" ,
4730         E = Ct ( V "F" * ( "," * V "F" ) ^ 0 ) ,
4731         F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4732             * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4733     }
4734     def_table = cut_definition : match ( definition )
4735 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.).

```

4736     local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4737     local tex_arg = tex_braced_arg + C ( 1 )
4738     local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4739     local args_for_tag
4740     = tex_option_arg
4741     * space ^ 0
4742     * tex_arg
4743     * space ^ 0
4744     * tex_arg
4745     local args_for_morekeywords
4746     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4747     * space ^ 0
4748     * tex_option_arg
4749     * space ^ 0
4750     * tex_arg
4751     * space ^ 0
4752     * ( tex_braced_arg + Cc ( nil ) )
4753     local args_for_moredelims
4754     = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4755     * args_for_morekeywords
4756     local args_for_morecomment
4757     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4758     * space ^ 0
4759     * tex_option_arg
4760     * space ^ 0
4761     * 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`.

```

4762     local sensitive = true
4763     local style_tag , left_tag , right_tag
4764     for _ , x in ipairs ( def_table ) do
4765         if x[1] == "sensitive" then
4766             if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4767                 sensitive = true
4768             else
4769                 if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4770             end
4771         end
4772         if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4773         if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4774         if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4775         if x[1] == "tag" then

```

```

4776     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4777     style_tag = style_tag or [[\PitonStyle{Tag}]]
4778 end
4779 end

```

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

```

4780 local Number =
4781   K ( 'Number.Internal' ,
4782     ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4783       + digit ^ 0 * "." * digit ^ 1
4784       + digit ^ 1 )
4785     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4786     + digit ^ 1
4787   )
4788 local string_extra_letters = ""
4789 for _ , x in ipairs ( extra_letters ) do
4790   if not ( extra_others[x] ) then
4791     string_extra_letters = string_extra_letters .. x
4792   end
4793 end
4794 local letter = alpha + S ( string_extra_letters )
4795   + P "â" + "à" + "ç" + "ê" + "ë" + "ê" + "ï" + "î"
4796   + "ô" + "û" + "ü" + "â" + "ã" + "ç" + "é" + "è" + "ê" + "ë"
4797   + "ï" + "î" + "ô" + "û" + "ü"
4798 local alphanum = letter + digit
4799 local identifier = letter * alphanum ^ 0
4800 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.

```

4801 local split_clist =
4802   P { "E" ,
4803     E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4804       * ( P "{" ) ^ 1
4805       * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4806       * ( P "}" ) ^ 1 * space ^ 0 ,
4807     F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4808   }

```

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

```

4809 local keyword_to_lpeg
4810 function keyword_to_lpeg ( name ) return
4811   Q ( Cmt (
4812     C ( identifier ) ,
4813     function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4814   )
4815   )
4816 end
4817 local Keyword = P ( false )
4818 local PrefixedKeyword = P ( false )

```

The variable `keywords_prefix` will be a string whose the characters will be the different values used with the key `keywordsprefix`.

```

4820 local keywords_prefix = ''

```

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

```

4821 for _ , x in ipairs ( def_table )
4822 do if x[1] == "morekeywords"
4823   or x[1] == "otherkeywords"
4824   or x[1] == "moredirectives"
4825   or x[1] == "moretexcs"
4826 then
4827   local keywords = P ( false )

```

```

4828     local style = [[\PitonStyle{Keyword}]]
4829     if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
4830     style = tex_option_arg : match ( x[2] ) or style
4831     local n = tonumber ( style )
4832     if n then
4833         if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4834     end

4835     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4836         if x[1] == "moretexcs" then
4837             keywords = Q ( [[\]] .. word ) + keywords
4838         else
4839             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*.

```

4840             then keywords = Q ( word ) + keywords
4841             else keywords = keyword_to_lpeg ( word ) + keywords
4842         end
4843     end
4844 end
4845 Keyword = Keyword +
4846     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4847 end

```

Of course, the feature with the key `keywordsprefix` is designed for the languages TeX, LaTeX, and *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.

```

4848     if x[1] == "keywordsprefix" then
4849         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4850         PrefixedKeyword = PrefixedKeyword
4851             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4852         keywords_prefix = keywords_prefix .. prefix
4853     end
4854 end

```

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

```

4855     local long_string = P ( false )
4856     local Long_string = P ( false )
4857     local LongString = P ( false )
4858     local central_pattern = P ( false )
4859     for _ , x in ipairs ( def_table ) do
4860         if x[1] == "morestring" then
4861             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4862             arg2 = arg2 or [[\PitonStyle{String.Long}]]
4863             if arg1 ~= "s" then
4864                 arg4 = arg3
4865             end
4866             central_pattern = 1 - S ( " \r" .. arg4 )
4867             if arg1 : match "b" then
4868                 central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4869             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...

```

4870             if arg1 : match "d" or arg1 == "m" then

```

```

4871     central_pattern = P ( arg3 .. arg3 ) + central_pattern
4872 end
4873 if arg1 == "m"
4874 then prefix = B ( 1 - letter - ")" - "]" )
4875 else prefix = P ( true )
4876 end

```

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

```

4877 long_string = long_string +
4878     prefix
4879     * arg3
4880     * ( space + central_pattern ) ^ 0
4881     * arg4

```

Now a pattern *with captures*.

```

4882 local pattern =
4883     prefix
4884     * Q ( arg3 )
4885     * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4886     * Q ( arg4 )

```

We will need Long_string in the nested comments.

```

4887 Long_string = Long_string + pattern
4888 LongString = LongString +
4889     Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4890     * pattern
4891     * Ct ( Cc "Close" )
4892 end
4893 end

```

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

```

4894 local braces = Compute_braces ( long_string )
4895 if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4896
4897 DetectedCommands =
4898     Compute_DetectedCommands ( lang , braces )
4899     + Compute_RawDetectedCommands ( lang , braces )
4900
4901 LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )

```

Now, we deal with the comments and the delims.

sCmment is for the comments (of morecomment) of type **s** (*standard*) or **n** (*nested*).

```

4902 local sComment = P ( false )

```

lComment is for the comments (of morecomment) of type **l** (*line*).

```

4903 local lComment = P ( false )

```

fCmment is for the comments (of morecomment) of type **f** (*first*).

```

4904 local fComment = P ( false )
4905 local fComment = P ( false )
4906
4907 for _ , x in ipairs ( def_table ) do
4908     if x[1] == "morecomment" then
4909         local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4910         arg2 = arg2 or [[\PitonStyle{Comment}]]

```

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

```

4911     if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4912     if arg1 : match "l" or arg1 : match "f" then
4913         local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4914         : match ( other_args )

```

```

4915 if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4916 if arg3 == [[\%]] then arg3 = "%" end -- mandatory
4917 local MyLPEG =
4918   Ct ( Cc "Open"
4919     * Cc ( "{" .. arg2 .. [[{\PitonSpaceSubstitute{}}] ]
4920     * Cc "}" }"
4921   )
4922   * Q ( arg3 )
4923   * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $ noqa
4924   * Ct ( Cc "Close" )
4925   * ( EOL + -1 )
4926 if arg1 : match "l" then
4927   lComment = lComment + MyLPEG
4928 else
4929   fComment = fComment + MyLPEG
4930 end
4931 else
4932   local arg3 , arg4 =
4933     ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
4934   if arg1 : match "s" then
4935     sComment = sComment +
4936       Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4937       * Q ( arg3 )
4938       * (
4939         CommentMath
4940         + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
4941         + EOL
4942       ) ^ 0
4943       * Q ( arg4 )
4944       * Ct ( Cc "Close" )
4945   end
4946   if arg1 : match "n" then
4947     sComment = sComment +
4948       Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4949       * P { "A" ,
4950         A = Q ( arg3 )
4951         * ( V "A"
4952           + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
4953             - S "\r$" ) ^ 1 ) -- $
4954           + long_string
4955           + "$" -- $
4956           * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
4957           * "$" -- $
4958           + EOL
4959         ) ^ 0
4960         * Q ( arg4 )
4961       }
4962       * Ct ( Cc "Close" )
4963   end
4964 end
4965 end

```

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

```

4966 if x[1] == "moredelim" then
4967   local arg1 , arg2 , arg3 , arg4 , arg5
4968   = args_for_moredelims : match ( x[2] )
4969   local MyFun = Q
4970   if arg1 == "*" or arg1 == "**" then
4971     function MyFun ( x )
4972       if x ~= '' then return
4973         LPEG1[lang] : match ( x )
4974       end
4975     end
4976   end

```

```

4977     local left_delim
4978     if arg2 : match "i" then
4979         left_delim = P ( arg4 )
4980     else
4981         left_delim = Q ( arg4 )
4982     end
4983     if arg2 : match "l" then
4984         sComment = sComment +
4985             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4986             * left_delim
4987             * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4988             * Ct ( Cc "Close" )
4989             * ( EOL + -1 )
4990     end
4991     if arg2 : match "s" then
4992         local right_delim
4993         if arg2 : match "i" then
4994             right_delim = P ( arg5 )
4995         else
4996             right_delim = Q ( arg5 )
4997         end
4998         sComment = sComment +
4999             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
5000             * left_delim
5001             * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
5002             * right_delim
5003             * Ct ( Cc "Close" )
5004     end
5005 end
5006 end
5007
5008 local Delim = Q ( S "{[()]}")
5009 local Punct = Q ( S "=",:;!\\'\"" )
5010
5010 local EOL =
5011     P "\r"
5012     *
5013     (
5014         space ^ 0 * -1
5015         +
5016         Cc "EOL"
5017     )
5018     * ( ( fComment + lComment ) ^ 0 * LeadingSpace ^ 0 * # ( 1 - S "\r" ) ) ^ -1

```

The following definition of the LPEG Word is added on 2026/07/01. Previously, the general definition of Word (defined at the beginning of the Lua part of the code of this extension piton) was used. The only difference is the term S(keywords_prefix).

```

5019     local lpeg_central = 1 - S " '\r[{}]" - digit - S ( keywords_prefix )
5020     if piton.begin_escape then
5021         lpeg_central = lpeg_central - piton.begin_escape
5022     end
5023     if piton.begin_escape_math then
5024         lpeg_central = lpeg_central - piton.begin_escape_math
5025     end
5026     local Word = Q ( lpeg_central ^ 1 )
5027
5027     local Main =
5028         space ^ 0 * EOL * ( fComment + lComment ) ^ 0
5029         + Space
5030         + Tab
5031         + Escape + EscapeMath
5032         + CommentLaTeX
5033         + Beamer
5034         + DetectedCommands

```

```

5035     + sComment
5036     + lComment

```

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

```

5037     + LongString
5038     + Delim
5039     + PrefixedKeyword
5040     + Keyword * ( -1 + # ( 1 - alphanum ) )
5041     + Punct
5042     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5043     + Number
5044     + 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.

```

5045     LPEG1[lang] = Main ^ 0

```

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

```

5046     LPEG2[lang] =
5047         Ct (
5048             ( space ^ 0 * P "\r" ) ^ -1
5049             * Lc [[ \@@_begin_line: ]]
5050             * ( fComment + lComment ) ^ 0 -- 2026-05-12
5051             * LeadingSpace ^ 0
5052             * ( space ^ 1 * -1 + Main ) ^ 0
5053             * -1
5054             * Lc [[ \@@_end_line: ]]
5055         )

```

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

```

5056     if left_tag then
5057         local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
5058             * Q ( left_tag * other ^ 0 ) -- $
5059             * ( ( 1 - P ( right_tag ) ) ^ 0 )
5060             / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
5061             * Q ( right_tag )
5062             * Ct ( Cc "Close" )
5063     MainWithoutTag
5064         = space ^ 1 * -1
5065         + space ^ 0 * EOL
5066         + Space
5067         + Tab
5068         + Escape + EscapeMath
5069         + CommentLaTeX
5070         + Beamer
5071         + DetectedCommands
5072         + sComment
5073         + Delim
5074         + LongString
5075         + PrefixedKeyword
5076         + Keyword * ( -1 + # ( 1 - alphanum ) )
5077         + Punct
5078         + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5079         + Number
5080         + Word
5081     LPEG0[lang] = MainWithoutTag ^ 0
5082     local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
5083                   + Beamer + DetectedCommands + sComment + Tag
5084     MainWithTag
5085         = space ^ 1 * -1
5086         + space ^ 0 * EOL
5087         + Space

```

```

5088         + LPEGaux
5089         + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
5090 LPEG1[lang] = MainWithTag ^ 0
5091 LPEG2[lang] =
5092   Ct (
5093     ( space ^ 0 * P "\r" ) ^ -1
5094     * Lc [[ \@@_begin_line: ]]
5095     * Beamer
5096     * LeadingSpace ^ 0
5097     * LPEG1[lang]
5098     * -1
5099     * Lc [[ \@@_end_line: ]]
5100   )
5101 end
5102 end

```

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

```

5103 function piton.write_files_now ( )
5104   for file_name , file_content in pairs ( piton.write_files ) do
5105     local file = io.open ( file_name , "w" )
5106     if file then
5107       file : write ( file_content )
5108       file : close ( )
5109     else
5110       sprintL3
5111       ( [[ \@@_error_or_warning:nn { FileError } { ]] .. file_name .. "]" )
5112     end
5113   end
5114 end

```

3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

5115 function piton.utf16 ( str )
5116   local hex = { "FEFF" } -- BOM UTF-16BE
5117   for _, codepoint in utf8.codes(str) do
5118     table.insert(hex, string.format("%04X", codepoint))
5119   end
5120   return table.concat(hex)
5121 end
5122 </LUA>

```