

The package `piton`*

F. Pantigny
fpantigny@wanadoo.fr

July 20, 2026

Abstract

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

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

1 Introduction

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

In fact, all that job is done by a LPEG called `LPEG1[<language>]` where `<language>` is a Lua string which is the name of the computer language. That LPEG, when matched against the string of a computer listing, returns as capture a Lua table containing data to send to LaTeX. The only thing to do after will be to apply `tex.tprint` to each element of that table.¹

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

Consider, for example, the following Python code:

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

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

*This document corresponds to the version 4.14a of `piton`, at the date of 2026/07/20.

¹Recall that `tex.tprint` takes in as argument a Lua table whose first component is a “catcode table” and the second element a string. The string will be sent to LaTeX with the regime of catcodes specified by the catcode table. If no catcode table is provided, the standard catcodes of LaTeX will be used.

```

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

```

^aEach line of the computer listings will be encapsulated in a pair: `\_@@_begin_line: – \_@@_end_line:`. The token `\_@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\_@@_begin_line:`. Both tokens `\_@@_begin_line:` and `\_@@_end_line:` will be nullified in the command `\piton` (since there can't be lines breaks in the argument of a command `\piton`).

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

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

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

```

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

```

2 The L3 part of the implementation

2.1 Declaration of the package

```

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

```

```

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

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

```

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

```

20 \RequirePackage { amstext }

```

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

```

21 \RequirePackage { marginalia }

```

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

```

22 \RequirePackage { transparent }

```

```

23 \AtBeginDocument
24 {
25   \IfPackageLoadedT { babel }
26     { \babelprovide { english } }
27 }

28 \cs_new_protected:Npn \@@_error:n { \msg_error:nn { piton } }
29 \cs_new_protected:Npn \@@_warning:n { \msg_warning:nn { piton } }
30 \cs_new_protected:Npn \@@_warning:nn { \msg_warning:nnn { piton } }
31 \cs_new_protected:Npn \@@_error:nn { \msg_error:nnn { piton } }
32 \cs_new_protected:Npn \@@_error:nnn { \msg_error:nnnn { piton } }
33 \cs_new_protected:Npn \@@_fatal:n { \msg_fatal:nn { piton } }
34 \cs_new_protected:Npn \@@_fatal:nn { \msg_fatal:nnn { piton } }
35 \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).

```

36 \cs_new_protected:Npn \@@_msg_new:nnn #1 #2 #3
37 {
38   \bool_if:NTF \g_@@_messages_for_Overleaf_bool
39     { \msg_new:nnn { piton } { #1 } { #2 } { #3 } }
40     { \msg_new:nnnn { piton } { #1 } { #2 } { #3 } }
41 }

```

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

```

42 \cs_new_protected:Npn \@@_error_or_warning:n
43 { \bool_if:NTF \g_@@_messages_for_Overleaf_bool \@@_warning:n \@@_error:n }
44 \cs_new_protected:Npn \@@_error_or_warning:nn
45 { \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".

```

46 \bool_new:N \g_@@_messages_for_Overleaf_bool
47 \bool_gset:Nn \g_@@_messages_for_Overleaf_bool
48 {

```

```

49     \str_if_eq_p:on \c_sys_jobname_str { _region_ } % for Emacs
50     || \str_if_eq_p:on \c_sys_jobname_str { output } % for Overleaf
51 }

52 \@@_msg_new:nn { LuaLaTeX-mandatory }
53 {
54     LuaLaTeX-is-mandatory.\
55     The~package~'piton'~requires~the~engine~LuaLaTeX.\
56     \str_if_eq:onT \c_sys_jobname_str { output }
57     { If~you~use~Overleaf,~you~can~switch~to~LuaLaTeX~in~
58       "File~>~Settings~>~Compiler"~and~if~you~use~TeXPage,
59       ~you~should~go~in~"Settings". \ }
60     \IfClassLoadedT { beamer }
61     {
62         Since~you~use~Beamer,~don't~forget~to~use~piton~in~frames~with~
63         the~key~'fragile'.\
64     }
65     \IfClassLoadedT { ltx-talk }
66     {
67         Since~you~use~'ltx-talk',~don't~forget~to~use~piton~in~
68         environments~'frame*'. \
69     }
70 }
71 \sys_if_engine_luatex:F { \@@_fatal:n { LuaLaTeX-mandatory } }

72 \RequirePackage { luacode }

73 \@@_msg_new:nnn { piton.lua-not-found }
74 {
75     The~file~'piton.lua'~can't~be~found.~
76     If~you~want~to~know~how~to~retrieve~the~file~'piton.lua',~type~H~<return>.
77 }
78 {
79     On~the~site~CTAN,~go~to~the~page~of~'piton':~https://ctan.org/pkg/piton.~
80     The~file~'README.md'~explains~how~to~retrieve~the~files~'piton.sty'~and~
81     'piton.lua'.
82 }

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

```

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

```

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

```

104 }

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

```

105 \ProcessKeyOptions

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

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

119 \RequirePackage { xcolor }

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

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

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

```

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

```

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

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

```

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

```

151 \IfClassLoadedTF { beamer }
152 { \bool_gset_false:N \g_@@_footnote_bool }
153 {
154     \IfPackageLoadedTF { footnote }

```

```

155         { \@@_error:n { footnotehyper~with~footnote~package } }
156         { \usepackage { footnotehyper } }
157     \bool_gset_true:N \g_@@_footnote_bool
158 }
159 }

```

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

```

160 \dim_new:N \l_@@_rounded_corners_dim
161 \bool_new:N \l_@@_in_label_bool
162 \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`).

```

163 \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:`).

```

164 \box_new:N \g_@@_output_box

```

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

```

165 \str_new:N \l_piton_language_str
166 \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.

```

167 \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`).

```

168 \seq_new:N \l_@@_path_seq

```

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

```

169 \seq_new:N \g_@@_join_seq

```

```

170 \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.

```

171 \dim_new:N \l_@@_tcb_margins_dim

```

The following parameter corresponds to the key `box`.

```

172 \str_new:N \l_@@_box_str

```

In order to have a better control over the keys.

```

173 \bool_new:N \l_@@_in_PitonOptions_bool
174 \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.

```

175 \int_new:N \g_@@_nb_lines_int

```

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

```

176 \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).

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

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

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

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

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

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

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

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

```
182 \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).

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

The argument of `\PitonInputFile`.

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

The following line can't be deleted.

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

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

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

```
187 \str_new:N \l_@@_paperclip_str
```

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

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

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

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

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

```
190 \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).

```
191 \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.

```
192 \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).

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

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

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`:

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

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

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

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

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

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

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

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

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

```
200 \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.

```
201 \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 `python`. 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.

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

```
203 \cs_new_protected:Npn \@@_tab:
```

```
204 {
```

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

```
206 {
```

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

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

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

```
    \(\mathcolor { gray }
```

```
        { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \)
```

```
212 }
```

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

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

```
215 }
```

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

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

When the key `break-lines-in-python` is set, that parameter will be replaced by `\space` (in `\python` 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.

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

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

```
218 \cs_new_protected:Npn \@@_label:n #1
```

```
219 {
```

```
    \bool_if:NTF \l_@@_line_numbers_bool
```

```
221 {
```

```
    \@bsphack
```

```
    \protected@write \@auxout { }
```

```
223 }
```

```

224     {
225         \string \newlabel { #1 }
226     {
227         { \int_use:N \g_@@_visual_line_int }
228         { \thepage }
229         { }
230         { line.#1 }
231         { }
232     }
233 }
234 \esphack
235 \IfPackageLoadedT { hyperref }
236 { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
237 }
238 { \@@_error:n { label-with-lines-numbers } }
239 }

```

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`.

```

240 \cs_new_protected:Npn \@@_zlabel:n #1
241 {
242     \bool_if:NTF \l_@@_line_numbers_bool
243     {
244         \@bsphack
245         \protected@write \@auxout { }
246         {
247             \string \zref@newlabel { #1 }
248             {
249                 \string \default { \int_use:N \g_@@_visual_line_int }
250                 \string \page { \thepage }
251                 \string \zc@type { line }
252                 \string \anchor { line.#1 }
253             }
254         }
255         \esphack
256         \IfPackageLoadedT { hyperref }
257         { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
258     }
259     { \@@_error:n { label-with-lines-numbers } }
260 }

```

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

```

261 \NewDocumentCommand { \@@_rowcolor:n } { o m }
262 {
263     \tl_gset:ce
264     { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
265     { \tl_if_novalue:nTF { #1 } { #2 } { [ #1 ] { #2 } } }
266     \bool_gset_true:N \g_@@_rowcolor_inside_bool
267 }

```

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).

```

268 \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.

```

269 \cs_new:Npn \@@_marker_beginning:n #1 { }
270 \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.

```
271 \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 replaced by `\@@_trailing_space:.`

```
272 \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`.

```
273 \bool_new:N \g_@@_color_is_none_bool
```

```
274 \bool_new:N \g_@@_next_color_is_none_bool
```

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

2.3 Detected commands

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

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

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

```
276 \clist_new:N \l_@@_detected_commands_clist
277 \clist_new:N \l_@@_raw_detected_commands_clist
278 \clist_new:N \l_@@_beamer_commands_clist
279 \clist_set:Nn \l_@@_beamer_commands_clist
280   { uncover , only , visible , invisible , alert , action }
281 \clist_new:N \l_@@_beamer_environments_clist
282 \clist_set:Nn \l_@@_beamer_environments_clist
283   { 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.

```
284 \hook_gput_code:nnn { begindocument } { . }
285   {
286     \newtoks \PitonDetectedCommands
287     \newtoks \PitonRawDetectedCommands
288     \newtoks \PitonBeamerCommands
289     \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.

```
290 \exp_args:NV \PitonDetectedCommands \l_@@_detected_commands_clist
291 \exp_args:NV \PitonRawDetectedCommands \l_@@_raw_detected_commands_clist
292 \exp_args:NV \PitonBeamerCommands \l_@@_beamer_commands_clist
293 \exp_args:NV \PitonBeamerEnvironments \l_@@_beamer_environments_clist
294 }
```

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.

```
295 \tl_new:N \g_@@_def_vertical_commands_tl
```

The following commands must *not* be protected.

```
296 \cs_new:Npn \@@_retrieve_backslash:n #1
297 {
298   \str_if_eq:eeTF \c_backslash_str { \str_head:n { #1 } }
299   { \str_tail:n { #1 } }
300   { #1 }
301 }
```

```
302 \cs_new_protected:Npn \@@_detected_commands:n #1
303 {
304   \clist_set:Nn \l_tmpa_clist { #1 }
305   \clist_clear:N \l_tmpb_clist
306   \clist_map_inline:Nn \l_tmpa_clist
307   { \clist_put_right:Ne \l_tmpb_clist { \@@_retrieve_backslash:n { ##1 } } }
308   \clist_if_in:NnTF \l_tmpb_clist { rowcolor }
309   {
310     \@@_error:n { rowcolor~in-detected-commands }
311     \clist_remove_all:Nn \l_tmpb_clist { rowcolor }
312   }
313   \clist_put_right:No \l_@@_detected_commands_clist \l_tmpb_clist
314 }
```

```
315 \cs_new_protected:Npn \@@_raw_detected_commands:n #1
316 {
317   \clist_set:Nn \l_tmpa_clist { #1 }
318   \clist_clear:N \l_tmpb_clist
319   \clist_map_inline:Nn \l_tmpa_clist
320   { \clist_put_right:Ne \l_tmpb_clist { \@@_retrieve_backslash:n { ##1 } } }
321   \clist_put_right:No \l_@@_raw_detected_commands_clist \l_tmpb_clist
322 }
```

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

```

343         { \tl_gput_right:cn }
344         { \tl_gset:cn }
345         { g_@@_after_line _ \int_eval:n { \g_@@_line_int + 1 }_tl }
346         { \use:c { @@_old _ ##1 : } { ####1 } }
347     }
348 }
349 \tl_gput_right:Nn \g_@@_def_vertical_commands_tl
350 { \cs_set_eq:cc { ##1 } { @@_new _ ##1 : n } }
351 }
352 }

```

2.4 Treatment of a line of code

```

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

```

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.

```

362         \bool_if:NT \l_@@_break_lines_in_Piton_bool
363         {
364             \tl_if_eq:NnF \l_@@_space_in_string_tl { }
365             { \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.

```

366         \tl_replace_all:Nvn \l_tmpa_tl
367         \c_catcode_other_space_tl
368         \@@_breakable_space:
369     }
370 }
371 \l_tmpa_tl
372 }
373 \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:`.

```
374 \cs_set_protected:Npn \@@_end_line: { }
```

```

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

```

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

```

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

```

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

```

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

```

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.

```

390           piton.empty_lines
391           [ \int_eval:n { \g_@@_line_int + 1 } ]
392         )
393       }
394     }
395     \bool_lazy_or:nnT
396     { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
397     { ! \l_@@_skip_empty_lines_bool }
398     { \int_gincr:N \g_@@_visual_line_int }
399
400     \bool_lazy_or:nnTF
401     { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
402     { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
403     {
404       \bool_lazy_or:nnTF
405       { \int_compare_p:nNn { \l_@@_numbers_step_int } = 1 }
406       {
407         \int_compare_p:nNn
408         {
409           \int_mod:nn
410           { \g_@@_visual_line_int }
411           { \l_@@_numbers_step_int }
412         }
413         = \c_one_int
414       }
415     }
416     {
417       \str_if_eq:eeTF \l_@@_line_numbers_position_str { left }
418       \@@_print_number_left:
419       {
420         \seq_gput_right:Ne \g_@@_visual_line_numbers_seq
421         { \int_use:N \g_@@_visual_line_int }
422       }
423     }
424     {
425       \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
426       { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
427     }
428   }
429   \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
430   { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
431 }
432

```

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).

```

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

438     \bool_if:NTF \l_@@_minimize_width_bool
439     {
440         \hbox_set:Nn \l_tmpa_box
441         {
442             \language = -1
443             \raggedright
444             \strut
445             \@@_replace_spaces:n { #1 }
446             \strut \hfil
447         }
448         \dim_compare:nNnTF { \box_wd:N \l_tmpa_box } < \l_@@_code_width_dim
449         { \box_use:N \l_tmpa_box }
450         { \@@_vtop_of_code:n { #1 } }
451     }
452     { \@@_vtop_of_code:n { #1 } }
453 }

```

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

```

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

```

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

```

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

```

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:.`

```

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

```

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`.

```
480     \group_begin:
481     \@@_compute_and_set_color:
```

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

```
482     \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
483     \bool_if:NT \g_@@_next_color_is_none_bool
484     { \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.

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

Now, the colored panel.

```
489     \dim_compare:nNnTF \l_@@_rounded_corners_dim > \c_zero_dim
490     {
491         \int_compare:nNnTF \g_@@_line_int = \c_one_int
492         {
493             \begin{tikzpicture}[baseline = 0cm]
494             \fill (0,0)
495                 [rounded-corners = \l_@@_rounded_corners_dim]
496                 -- (0,\l_@@_tmpc_dim)
497                 -- (\l_tmpb_dim,\l_@@_tmpc_dim)
498                 [sharp-corners] -- (\l_tmpb_dim,-\l_tmpa_dim)
499                 -- (0,-\l_tmpa_dim)
500                 -- cycle ;
501             \end{tikzpicture}
502         }
503         {
504             \int_compare:nNnTF \g_@@_line_int = \g_@@_nb_lines_int
505             {
506                 \begin{tikzpicture}[baseline = 0cm]
507                 \fill (0,0) -- (0,\l_@@_tmpc_dim)
508                     -- (\l_tmpb_dim,\l_@@_tmpc_dim)
509                     [rounded-corners = \l_@@_rounded_corners_dim]
510                     -- (\l_tmpb_dim,-\l_tmpa_dim)
511                     -- (0,-\l_tmpa_dim)
512                     -- cycle ;
513                 \end{tikzpicture}
514             }
515             {
516                 \vrule height \l_@@_tmpc_dim
517                 depth \l_tmpa_dim
518                 width \l_tmpb_dim
```

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

```
519         \dim_compare:nNnTF \l_tmpb_dim = \c_zero_dim
520         { \skip_horizontal:N \l_@@_listing_width_dim }
521     }
522 }
523 }
524 {
525     \vrule height \l_@@_tmpc_dim
526     depth \l_tmpa_dim
527     width \l_tmpb_dim
```

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

```
528     \dim_compare:nNnTF \l_tmpb_dim = \c_zero_dim
529     { \skip_horizontal:N \l_@@_listing_width_dim }
```

```

530         }
The group is for the color of the background.
531     \group_end:
532     \bool_if:NT \l_@@_line_numbers_bool
533     {
534         \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
535         {
536             \seq_gpop_right:NN \g_@@_visual_line_numbers_seq \l_tmpa_tl
537             \@@_print_number_right:
538         }
539     }
540 }
541 \bool_if:NT \g_@@_next_color_is_none_bool
542 { \skip_vertical:n { 2.5 pt } }
543 \skip_vertical:n { - \box_ht_plus_dp:N \l_@@_line_box }
544 \box_use_drop:N \l_@@_line_box
545 }
546 }

```

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.

```

547 \cs_set_protected:Npn \@@_compute_and_set_color:
548 {
549     \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
550     { \tl_set:Nn \l_tmpa_tl { none } }
551     {
552         \int_set:Nn \l_tmpb_int
553         { \int_mod:nn \g_@@_line_int \l_@@_bg_colors_int + 1 }
554         \tl_set:Ne \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
555     }

```

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

```

556 \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
557 {
558     \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).

```

559     \cs_undefine:c { g_@@_color_ \int_use:N \g_@@_line_int _ tl}
560 }
561 \tl_if_eq:NnTF \l_tmpa_tl { none }
562 { \bool_gset_true:N \g_@@_color_is_none_bool }
563 {
564     \bool_gset_false:N \g_@@_color_is_none_bool
565     \@@_color:o \l_tmpa_tl
566 }

```

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).

```

567 \int_compare:nNnTF { \g_@@_line_int + 1 } = \g_@@_nb_lines_int
568 { \bool_gset_false:N \g_@@_next_color_is_none_bool }
569 {
570     \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
571     { \tl_set:Nn \l_tmpa_tl { none } }
572     {
573         \int_set:Nn \l_tmpb_int
574         { \int_mod:nn { \g_@@_line_int + 1 } \l_@@_bg_colors_int + 1 }
575         \tl_set:Ne \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
576     }
577     \cs_if_exist:cT { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
578     {

```

```

579         \tl_set_eq:Nc \l_tmpa_tl
580         { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
581     }
582     \tl_if_eq:NnTF \l_tmpa_tl { none }
583     { \bool_gset_true:N \g_@@_next_color_is_none_bool }
584     { \bool_gset_false:N \g_@@_next_color_is_none_bool }
585 }
586 }

```

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} }`.

```

587 \cs_set_protected:Npn \@@_color:n #1
588 {
589     \tl_if_head_eq_meaning:nNTF { #1 } [
590     {
591         \tl_set:Nn \l_tmpa_tl { #1 }
592         \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
593         \exp_last_unbraced:No \color \l_tmpa_tl
594     }
595     { \color { #1 } }
596 }
597 \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.

```

598 \cs_new_protected:Npn \@@_par:
599 {

```

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

```

600     \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.

```

601     \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.

```

602     \kern -2.5 pt

```

Now, we control page breaks after the paragraph.

```

603     \@@_add_penalty_for_the_line:
604 }

```

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`.

```

605 \cs_set_protected:Npn \@@_breakable_space:
606 {
607     \discretionary
608     { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
609     {
610         \hbox_overlap_left:n
611         {
612             {

```

```

613         \normalfont \footnotesize \color { gray }
614         \l_@@_continuation_symbol_tl
615     }
616     \skip_horizontal:n { 0.3 em }
617     \int_compare:nNnT \l_@@_bg_colors_int > \c_zero_int
618     { \skip_horizontal:n { 0.5 em } }
619 }
620 \bool_if:NT \l_@@_indent_broken_lines_bool
621 {
622     \hbox:n
623     {
624         \prg_replicate:nn { \g_@@_indentation_int } { ~ }
625         { \color { gray } \l_@@_csoi_tl }
626     }
627 }
628 }
629 { \hbox { ~ } }
630 }

```

2.5 PitonOptions

```

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

646 \keys_define:nn { PitonOptions / marker }
647 {
648     beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
649     beginning .value_required:n = true ,
650     end .cs_set:Np = \@@_marker_end:n #1 ,
651     end .value_required:n = true ,
652     include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
653     unknown .code:n = \@@_error:n { Unknown-key-for-marker }
654 }

655 \keys_define:nn { PitonOptions / line-numbers }
656 {
657     true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
658     false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
659
660     start .code:n =
661         \bool_set_true:N \l_@@_line_numbers_bool
662         \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
663     start .value_required:n = true ,
664
665     skip-empty-lines .code:n =
666         \bool_if:NF \l_@@_in_PitonOptions_bool
667         { \bool_set_true:N \l_@@_line_numbers_bool }
668     \str_if_eq:nnTF { #1 } { false }

```

```

669         { \bool_set_false:N \l_@@_skip_empty_lines_bool }
670         { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
671
672     label-empty-lines .code:n =
673         \bool_if:NF \l_@@_in_PitonOptions_bool
674         { \bool_set_true:N \l_@@_line_numbers_bool }
675         \str_if_eq:nnTF { #1 } { false }
676         { \bool_set_false:N \l_@@_label_empty_lines_bool }
677         { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
678
679     absolute .code:n =
680         \bool_if:NTF \l_@@_in_PitonOptions_bool
681         { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
682         { \bool_set_true:N \l_@@_line_numbers_bool }
683         \bool_if:NT \l_@@_in_PitonInputFile_bool
684         {
685             \bool_set_true:N \l_@@_line_numbers_absolute_bool
686             \bool_set_false:N \l_@@_skip_empty_lines_bool
687         } ,
688     absolute .value_forbidden:n = true ,
689
690     resume .code:n =
691         \bool_set_true:N \l_@@_resume_bool
692         \bool_if:NF \l_@@_in_PitonOptions_bool
693         { \bool_set_true:N \l_@@_line_numbers_bool } ,
694     resume .value_forbidden:n = true ,
695
696     sep .dim_set:N = \l_@@_numbers_sep_dim ,
697     sep .value_required:n = true ,
698     sep .initial:n = 0.7 em ,
699
700     step .int_set:N = \l_@@_numbers_step_int ,
701     step .initial:n = 1 ,
702     step .value_required:n = true ,
703
704     position .choices:nn = { left , right }
705     { \str_set_eq:NN \l_@@_line_numbers_position_str \l_keys_choice_tl } ,
706     position .value_required:n = true ,
707
708     format .tl_set:N = \l_@@_line_numbers_format_tl ,
709     format .value_required:n = true ,
710
711     format+ .code:n = \tl_put_right:Nn \l_@@_line_numbers_format_tl { #1 } ,
712     format+ .value_required:n = true ,
713
714     format~+ .meta:n = { format+ = #1 } ,
715
716     lmmono10-drawn .bool_set:N = \l_@@_lmodern_drawn_bool ,
717     lmmono10-drawn .default:n = true ,
718
719     unknown .code:n =
720         \@@_unknown_key:nn
721         { PitonOptions / line-numbers }
722         { Unknown~key~for~line-numbers }
723
724 }

```

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

```

725 \keys_define:nn { PitonOptions }
726 {
727     indentations-for-Foxit .choices:nn = { true , false }
728     {

```

```

729     \tl_if_eq:VnTF \l_keys_value_tl { true }
730     { \@@_define_leading_space_Foxit: }
731     { \@@_define_leading_space_normal: }
732   } ,
733   box .choices:nn = { c , t , b , m }
734   { \str_set_eq:NN \l_@@_box_str \l_keys_choice_tl } ,
735   box .default:n = c ,
736   break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
737   break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,

```

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

```

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

```

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

```

757   begin-escape .code:n =
758     \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
759   begin-escape .value_required:n = true ,
760   begin-escape .usage:n = preamble ,
761
762   after-begin-escape .tl_set:N = \l_@@_after_begin_escape_tl ,
763   after-begin-escape .value_required:n = true ,
764
765   end-escape .code:n =
766     \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
767   end-escape .value_required:n = true ,
768   end-escape .usage:n = preamble ,
769
770   before-end-escape .tl_set:N = \l_@@_before_end_escape_tl ,
771   before-end-escape .value_required:n = true ,
772
773   begin-escape-math .code:n =
774     \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
775   begin-escape-math .value_required:n = true ,
776   begin-escape-math .usage:n = preamble ,
777
778   end-escape-math .code:n =
779     \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
780   end-escape-math .value_required:n = true ,
781   end-escape-math .usage:n = preamble ,
782
783   comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
784   comment-latex .value_required:n = true ,
785   comment-latex .usage:n = preamble ,
786
787   label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
788   label-as-zlabel .usage:n = preamble ,
789

```

```

790 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
791 math-comments .usage:n = preamble ,

```

Now, general keys.

```

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

```

803 path .initial:n = . ,
804 path-write .str_set:N = \l_@@_path_write_str ,
805 path-write .value_required:n = true ,
806 font-command .tl_set:N = \l_@@_font_command_tl ,
807 font-command .initial:n = \ttfamily ,
808 font-command .value_required:n = true ,
809 font-command+ .code:n
810 = { \tl_put_right:Nn \l_@@_font_command_tl { #1 } } ,
811 font-command+ .value_required:n = true ,
812 font-command~+ .meta:n = { font-command+ = #1 } ,
813 font-command~+ .value_required:n = true ,
814 gobble .int_set:N = \l_@@_gobble_int ,
815 gobble .default:n = -1 ,
816 auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
817 auto-gobble .value_forbidden:n = true ,
818 env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
819 env-gobble .value_forbidden:n = true ,
820 tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
821 tabs-auto-gobble .value_forbidden:n = true ,
822
823 splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
824
825 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.

```

826 split-separation .tl_set:N = \l_@@_split_separation_tl ,
827 split-separation .value_required:n = true ,
828 split-separation .initial:n = \vspace { \baselineskip } \vspace { -1.25pt } ,
829
830 split-separation+ .code:n =
831   \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
832 split-separation+ .value_required:n = true ,
833 split-separation~+ .meta:n = { split-separation+ = #1 } ,
834 add-to-split-separation .meta:n = { split-separation+ = #1 } ,
835
836 marker .code:n =
837   \bool_lazy_or:nnTF
838     \l_@@_in_PitonInputFile_bool
839     \l_@@_in_PitonOptions_bool
840     { \keys_set:nn { PitonOptions / marker } { #1 } }
841     { \@@_error:n { Invalid~key } } ,
842 marker .value_required:n = true ,
843
844 line-numbers .code:n =

```

```
845 \keys_set:nn { PitonOptions / line-numbers } { #1 } ,
```

The following line is mandatory.

```
846 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).

```
847 splittable .int_set:N = \l_@@_splittable_int ,
848 splittable .default:n = 1 ,
849 splittable .initial:n = 100 ,
850 background-color .code:n =
851 \clist_set:Nn \l_@@_bg_color_clist { #1 }
```

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

```
852 \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
853 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`).

```
854 prompt-background-color .tl_set:N = \l_@@_prompt_bg_color_tl ,
855 prompt-background-color .value_required:n = true ,
856 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.

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

```

894 join .value_required:n = true ,
895 join-separation .str_set:N = \l_@@_join_separation_str ,
896 join-separation .value_required:n = true ,
897 no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,
898 no-join .value_forbidden:n = true ,
899
900 left-margin .code:n =
901   \str_if_eq:nnTF { #1 } { auto }
902   {
903     \dim_zero:N \l_@@_left_margin_dim
904     \bool_set_true:N \l_@@_left_margin_auto_bool
905   }
906   {
907     \dim_set:Nn \l_@@_left_margin_dim { #1 }
908     \bool_set_false:N \l_@@_left_margin_auto_bool
909   } ,
910 left-margin .value_required:n = true ,
911
912 right-margin .code:n =
913   \str_if_eq:nnTF { #1 } { auto }
914   {
915     \dim_zero:N \l_@@_right_margin_dim
916     \bool_set_true:N \l_@@_right_margin_auto_bool
917   }
918   {
919     \dim_set:Nn \l_@@_right_margin_dim { #1 }
920     \bool_set_false:N \l_@@_right_margin_auto_bool
921   } ,
922 right-margin .value_required:n = true ,
923
924 tab-size .int_set:N = \l_@@_tab_size_int ,
925 tab-size .value_required:n = true ,
926 tab-size .initial:n = 4 ,
927
928 show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
929 show-spaces .value_forbidden:n = true ,
930
931 show-spaces-in-strings .code:n =
932   \tl_set:Nn \l_@@_space_in_string_tl { \ } , % U+2423
933 show-spaces-in-strings .value_forbidden:n = true ,
934
935 break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
936 break-lines-in-Piton .initial:n = true ,
937
938 break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
939
940 break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
941 break-lines .value_forbidden:n = true ,
942
943 indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,
944
945 end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
946 end-of-broken-line .value_required:n = true ,
947 end-of-broken-line .initial:n = \hspace* { 0.5em } \textbackslash ,
948
949 continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
950 continuation-symbol .value_required:n = true ,
951 continuation-symbol .initial:n = + ,
952
953 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
954 continuation-symbol-on-indentation .value_required:n = true ,
955 continuation-symbol-on-indentation .initial:n = $\hookrightarrow$ ,
956

```

```

957 first-line .code:n = \@@_in_PitonInputFile:n
958 { \int_set:Nn \l_@@_first_line_int { #1 } } ,
959 first-line .value_required:n = true ,
960
961 last-line .code:n = \@@_in_PitonInputFile:n
962 { \int_set:Nn \l_@@_last_line_int { #1 } } ,
963 last-line .value_required:n = true ,
964
965 begin-range .code:n = \@@_in_PitonInputFile:n
966 { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
967 begin-range .value_required:n = true ,
968
969 end-range .code:n = \@@_in_PitonInputFile:n
970 { \str_set:Nn \l_@@_end_range_str { #1 } } ,
971 end-range .value_required:n = true ,
972
973 range .code:n = \@@_in_PitonInputFile:n
974 {
975   \str_set:Nn \l_@@_begin_range_str { #1 }
976   \str_set:Nn \l_@@_end_range_str { #1 }
977 } ,
978 range .value_required:n = true ,
979
980 env-used-by-split .code:n =
981   \lua_now:n { piton.env_used_by_split = '#1' } ,
982 env-used-by-split .initial:n = Piton ,
983
984 resume .meta:n = line-numbers/resume ,
985
986 unknown .code:n =
987   \@@_unknown_key:nn
988   { PitonOptions }
989   { Unknown~key~for~PitonOptions } ,
990
991 % deprecated
992 all-line-numbers .code:n =
993   \bool_set_true:N \l_@@_line_numbers_bool
994   \bool_set_false:N \l_@@_skip_empty_lines_bool ,
995
996 rounded-corners .code:n =
997   \AtBeginDocument
998   {
999     \IfPackageLoadedTF { tikz }
1000     { \dim_set:Nn \l_@@_rounded_corners_dim { #1 } }
1001     { \@@_err_rounded_corners_without_Tikz: }
1002   } ,
1003   rounded-corners .default:n = 4 pt
1004 }
1005
1006 \hook_gput_code:nnn { begindocument } { . }
1007 {
1008   \IfPackageLoadedTF { tcolorbox }
1009   {
1010     \pgfkeysifdefined { / tcb / libload / breakable }
1011     {
1012       \keys_define:nn { PitonOptions }
1013       {
1014         tcolorbox .bool_set:N = \l_@@_tcolorbox_bool ,
1015       }
1016     }
1017     {
1018       \keys_define:nn { PitonOptions }
1019       { tcolorbox .code:n = \@@_error:n { library-breakable-not-loaded } }
1020     }
1021   }
1022 }

```

```

1020     {
1021       \keys_define:nn { PitonOptions }
1022       { tcolorbox .code:n = \@@_error:n { tcolorbox~not~loaded } }
1023     }
1024   }

1025 \cs_new_protected:Npn \@@_err_rounded_corners_without_Tikz:
1026   {
1027     \@@_error:n { rounded-corners~without~Tikz }
1028     \cs_gset:Npn \@@_err_rounded_corners_without_Tikz: { }
1029   }

1030 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
1031   {
1032     \bool_if:NTF \l_@@_in_PitonInputFile_bool
1033       { #1 }
1034       { \@@_error:n { Invalid-key } }
1035   }

1036 \NewDocumentCommand \PitonOptions { m }
1037   {
1038     \bool_set_true:N \l_@@_in_PitonOptions_bool
1039     \keys_set:nn { PitonOptions } { #1 }
1040     \bool_set_false:N \l_@@_in_PitonOptions_bool
1041   }

```

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.

```

1042 \NewDocumentCommand \@@_fake_PitonOptions { }
1043   { \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.

```

1044 \int_new:N \g_@@_visual_line_int
1045 \cs_new_protected:Npn \@@_incr_visual_line:
1046   { \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`.

```

1047 \cs_new_protected:Npn \@@_print_number_left:
1048   {
1049     \hbox_overlap_left:n
1050     {
1051       \@@_actually_print_number:n { \int_to_arabic:n { \g_@@_visual_line_int } }
1052       \skip_horizontal:N \l_@@_numbers_sep_dim
1053     }
1054   }

```

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`.

```

1055 \cs_new_protected:Npn \@@_print_number_right:
1056   {
1057     \hbox_overlap_left:n

```

```

1058 {
1059   \@@_actually_print_number:n { \l_tmpa_tl }
1060   \int_compare:nNnT \l_@@_bg_colors_int > 0
1061     { \skip_horizontal:n { 0.1 em } }
1062 }
1063 }

```

`\@@_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:`:

```

1064 \cs_new_protected:Npn \@@_actually_print_number:n #1
1065 {
1066   \group_begin:
1067   \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`.

```

1068   { \l_@@_line_numbers_format_tl { \@@_draw_number:e { #1 } } }
1069   {

```

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

```

1070     \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
1071     \l_@@_line_numbers_format_tl { #1 }
1072     \pdfextension literal { EMC }
1073   }
1074   \group_end:
1075 }

```

2.7 The main commands and environments for the end user

```

1076 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
1077 {
1078   \tl_if_novalue:nTF { #3 }

```

The last argument is provided by curryfication.

```

1079   { \@@_NewPitonLanguage:nnn { #1 } { #2 } }

```

The two last arguments are provided by curryfication.

```

1080   { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
1081 }

```

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.

```

1082 \prop_new:N \g_@@_languages_prop

```

```

1083 \keys_define:nn { NewPitonLanguage }
1084 {
1085   morekeywords .code:n = ,
1086   otherkeywords .code:n = ,
1087   sensitive .code:n = ,
1088   keywordsprefix .code:n = ,
1089   moretexcs .code:n = ,
1090   morestring .code:n = ,
1091   morecomment .code:n = ,
1092   moredelim .code:n = ,
1093   moredirectives .code:n = ,
1094   tag .code:n = ,
1095   alsodigit .code:n = ,
1096   alsoletter .code:n = ,
1097   alsoother .code:n = ,
1098   unknown .code:n = \@@_error:n { Unknown~key~NewPitonLanguage }
1099 }

```

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

```
1100 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
1101 {
```

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}{...}`.

```
1102   \tl_set:Nx \l_tmpa_tl
1103   {
1104     \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
1105     \str_lowercase:n { #2 }
1106   }
```

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

```
1107   \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.

```
1108   \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).

```
1109   \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
1110 }
1111 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
1112 {
1113   \hook_gput_code:nnn { begindocument } { . }
1114   { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }
1115 }
1116 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }
```

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

```
1117 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4 #5
1118 {
```

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}{...}`

```
1119   \tl_set:Nx \l_tmpa_tl
1120   {
1121     \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
1122     \str_lowercase:n { #4 }
1123   }
```

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`.

```
1124   \prop_get:NoNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl
```

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

```
1125   { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
1126   { \@@_error:n { Language~not~defined } }
1127 }
```

```
1128 \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.

```
1129   { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
1130   \cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }
```

```
1131 \NewDocumentCommand { \piton } { }
1132 { \peek_meaning:NTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }
1133 \NewDocumentCommand { \@@_piton_standard } { m }
1134 {
```

```

1135 \group_begin:
1136 \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
1137 {

```

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.

```

1138 \bool_lazy_or:nnT
1139 \l_@@_break_lines_in_piton_bool
1140 \l_@@_break_strings_anywhere_bool
1141 { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
1142 }

```

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

```

1143 \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:

```

1144 \cs_set_eq:NN \_ \c_backslash_str
1145 \cs_set_eq:NN \% \c_percent_str
1146 \cs_set_eq:NN \{ \c_left_brace_str
1147 \cs_set_eq:NN \} \c_right_brace_str
1148 \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.

```

1149 \cs_set_eq:cN { ~ } \space
1150 \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1151 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1152 \tl_set:Ne \l_tmpa_tl
1153 {
1154 \lua_now:e
1155 { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1156 { #1 }
1157 }
1158 \bool_if:NTF \l_@@_show_spaces_bool
1159 { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1160 {
1161 \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.

```

1162 { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl \space }
1163 }

```

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

```

1164 \if_mode_math:
1165 \text { \l_@@_font_command_tl \l_tmpa_tl }
1166 \else:
1167 \l_@@_font_command_tl \l_tmpa_tl
1168 \fi:
1169 \group_end:
1170 }

```

```

1171 \NewDocumentCommand { \@@_piton_verbatim } { v }
1172 {
1173 \group_begin:
1174 \automatichyphenmode = 1
1175 \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1176 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1177 \tl_set:Ne \l_tmpa_tl

```

```

1178     {
1179         \lua_now:e
1180         { piton.Parse('\l_piton_language_str',token.scan_string()) }
1181         { #1 }
1182     }
1183     \bool_if:NT \l_@@_show_spaces_bool
1184     { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { } } % U+2423
1185     \if_mode_math:
1186         \text { \l_@@_font_command_tl \l_tmpa_tl }
1187     \else:
1188         \l_@@_font_command_tl \l_tmpa_tl
1189     \fi:
1190     \group_end:
1191 }

```

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).

```

1192 \cs_new_protected:Npn \@@_piton:n #1
1193 { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }
1194
1195 \cs_new_protected:Npn \@@_piton_i:n #1
1196 {
1197     \group_begin:
1198     \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
1199     \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1200     \cs_set:cpn { pitonStyle _ Prompt } { }
1201     \cs_set_eq:NN \@@_leading_space: \space
1202     \cs_set_eq:NN \@@_trailing_space: \space
1203     \tl_set:Nx \l_tmpa_tl
1204     {
1205         \lua_now:e
1206         { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1207         { #1 }
1208     }
1209     \bool_if:NT \l_@@_show_spaces_bool
1210     { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { } } % U+2423
1211     \@@_replace_spaces:o \l_tmpa_tl
1212     \group_end:
1213 }

```

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

```

1214 \cs_new_protected:Npn \@@_pre_composition:
1215 {
1216     \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
1217     {
1218         \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).

```

1219         \str_if_empty:NF \l_@@_box_str
1220         { \bool_set_true:N \l_@@_minimize_width_bool }
1221     }

```

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).

```

1222     \dim_set:Nn \l_@@_listing_width_dim
1223     {
1224         \bool_if:NTF \l_@@_tcolorbox_bool

```

```

1225     {
1226         \l_@@_width_dim -
1227         ( \kvtcb@left@rule
1228         + \kvtcb@leftupper
1229         + \kvtcb@boxsep * 2
1230         + \kvtcb@rightupper
1231         + \kvtcb@right@rule )
1232     }
1233     { \l_@@_width_dim }
1234 }
1235 \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1236 \automatichyphenmode = 1
1237 \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1238 \g_@@_def_vertical_commands_tl
1239 \int_gzero:N \g_@@_line_int
1240 \int_gzero:N \g_@@_nb_lines_int
1241 \dim_zero:N \parindent
1242 \dim_zero:N \lineskip
1243 \dim_zero:N \parskip
1244
1245 \seq_gclear:N \g_@@_visual_line_numbers_seq
1246
1247 \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

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

```

1248 \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1249 { \bool_set_true:N \l_@@_bg_bool }
1250 \bool_gset_false:N \g_@@_rowcolor_inside_bool
1251 \IfPackageLoadedTF { zref-base }
1252 {
1253     \bool_if:NTF \g_@@_label_as_zlabel_bool
1254     { \cs_set_eq:NN \label \@@_zlabel:n }
1255     { \cs_set_eq:NN \label \@@_label:n }
1256     \cs_set_eq:NN \zlabel \@@_zlabel:n
1257 }
1258 { \cs_set_eq:NN \label \@@_label:n }
1259 \l_@@_font_command_tl
1260 }

```

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`.

```

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

```

```

1279         { \g_@@_nb_lines_int }
1280     }
1281 }
1282 \dim_set:Nn #1 { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1283 }

```

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.

```

1284 \cs_new_protected:Npn \@@_compute_code_width:
1285 {
1286     \dim_set:Nn \l_@@_code_width_dim
1287     {
1288         \l_@@_listing_width_dim
1289         -
1290         (
1291             \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1292             {
1293                 \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1294                 { \l_@@_left_margin_dim }
1295                 { 0.5 em }
1296             }
1297             +
1298             \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1299             { \l_@@_right_margin_dim }
1300             { 0.5 em }
1301         )
1302     }
1303 }
1304 }

```

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:`.

```

1305 \cs_new_protected:Npn \@@_recompute_listing_width:
1306 {
1307     \dim_set:Nn \l_@@_listing_width_dim
1308     {
1309         \box_wd:N \g_@@_output_box
1310         +
1311         \int_compare:nNnTF \l_@@_bg_colors_int > \c_zero_int
1312         {
1313             \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1314             { \l_@@_left_margin_dim }
1315             { 0.5 em }
1316         }
1317         +
1318         \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1319         { \l_@@_right_margin_dim }
1320         { 0.5 em }
1321     }
1322 }
1323 }

```

```

1324 \cs_new_protected:Npn \@@_store_body:n #1
1325 {

```

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

```

1326 \tl_set:Nc \obeyedline { \char_generate:nn { 13 } { 11 } }
1327 \tl_set:Nc \l_@@_listing_tl { #1 }
1328 \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1329 }

```

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

```

1330 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nnnnn
1331 {
1332   \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1333   {
1334
1335     \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1336     #4
1337     \@@_pre_composition:
1338     \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1339     {
1340       \int_gset:Nn \g_@@_visual_line_int
1341       { \l_@@_number_lines_start_int - 1 }
1342     }
1343     \bool_if:NT \g_@@_beamer_bool
1344     { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1345     \bool_if:NT \g_@@_footnote_bool \savenotes
1346     \IfPackageLoadedT { babel }
1347     { \begin { otherlanguage } { english } }
1348     \@@_composition:
1349     \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1350     { \@@_create_paperclip_or_annotation: }
1351     \IfPackageLoadedT { babel } { \end { otherlanguage } }
1352     \bool_if:NT \g_@@_footnote_bool \endsavenotes
1353     #5
1354   }
1355   { \ignorespacesafterend }
1356 }

```

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

```

1357 \cs_new_protected:Npn \@@_create_paperclip_or_annotation:
1358 {
1359   \marginalia [ pos = right ]
1360   {
1361     \vspace* { - 0.8 em }
1362     \hbox:n
1363     {
1364       \vrule-height~0~pt~depth~12~pt~width~0~pt
1365       \bool_if:NT \l_@@_annotation_bool
1366       {
1367         \lua_now:n
1368         {

```

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!

```

1369         pdf.immediateobj
1370         ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1371       }
1372     \pdfextension annot~width~5pt~height~10pt~depth~0pt
1373     {
1374       /Subtype /Text
1375       /Contents~\pdf_object_ref_last:
1376       /Name /Note
1377       /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).

```

1378             /ReplyType /Group
Adds the bit 10 which means LockedContents.
1379             /F~512
1380             /C [0.8~0.8~0.8]
1381         }
1382         \hspace* { 7 mm }
1383     }
1384     \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1385 }
1386 }
1387 }

1388 \cs_new_protected:Npn \@@_create_paperclip:
1389 {
1390     \str_if_empty:NT \l_@@_paperclip_str
1391     {
1392         \int_gincr:N \g_@@_paperclip_int
1393         \str_set:Ne \l_@@_paperclip_str { listing\_int\_use:N \g_@@_paperclip_int .txt }
1394     }

```

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

```

1395     \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1396     \box_move_down:nn
1397     { 10 pt }
1398     {
1399         \hbox:n
1400         {
1401             \pdfextension annot~width~10pt~height~20pt~depth~0pt
1402             {
1403                 /Subtype /FileAttachment
1404                 /Name /Paperclip
1405                 /F~8 % no zoom

```

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

```

1406             /Contents (The~computer~listing)
1407             /FS <<
1408                 /Type /Filespec
1409                 /F (\l_@@_paperclip_str)
1410                 /EF << /F~\pdf_object_ref_last: >>
1411                 /AFRelationship /Supplement
1412             >>
1413         }
1414     }
1415 }
1416 }

```

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

```

1417 \NewDocumentCommand { \NewPitonEnvironment } { }
1418 { \@@_DefinePitonEnvironment:nnnnn { New } }
1419 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1420 { \@@_DefinePitonEnvironment:nnnnn { Declare } }
1421 \NewDocumentCommand { \RenewPitonEnvironment } { }
1422 { \@@_DefinePitonEnvironment:nnnnn { Renew } }
1423 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1424 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

1425 \cs_new_protected:Npn \@@_translate_beamer_env:n
1426 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1427 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1428 \cs_new_protected:Npn \@@_composition:
1429 {

```

```

1430 \str_if_empty:NT \l_@@_box_str
1431 {
1432   \mode_if_vertical:F
1433   { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1434 }
1435 \bool_if:NT \l_@@_line_numbers_bool
1436 {
1437   \bool_lazy_and:nnT
1438   { \l_@@_left_margin_auto_bool }
1439   { \str_if_eq_p:ee \l_@@_line_numbers_position_str { left } }
1440   { \@@_compute_margin:N \l_@@_left_margin_dim }
1441   \bool_lazy_and:nnT
1442   { \l_@@_right_margin_auto_bool }
1443   { \str_if_eq_p:ee \l_@@_line_numbers_position_str { right } }
1444   { \@@_compute_margin:N \l_@@_right_margin_dim }
1445 }
1446 \lua_now:e
1447 {
1448   piton.join_separation = "\l_@@_join_separation_str"
1449   piton.join = "\l_@@_join_str"
1450   piton.write = "\l_@@_write_str"
1451   piton.path_write = "\l_@@_path_write_str"
1452 }
1453 \noindent
1454 \bool_if:NTF \l_@@_print_bool
1455 {

```

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.

```

1456 \bool_if:NTF \l_@@_split_on_empty_lines_bool
1457 { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1458 {
1459   \@@_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).

```

1460 \bool_if:NTF \l_@@_tcolorbox_bool
1461 {
1462   \str_if_empty:NTF \l_@@_box_str
1463   \@@_composition_iii:
1464   \@@_composition_iv:
1465 }
1466 {
1467   \str_if_empty:NTF \l_@@_box_str
1468   \@@_composition_i:
1469   \@@_composition_ii:
1470 }
1471 }
1472 }
1473 { \@@_gobble_parse_no_print:o \l_@@_listing_tl }
1474 }

```

`\@@_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=...`

```

1475 \cs_new_protected:Npn \@@_composition_i:
1476 {

```

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.

```
1477 \box_clear:N \g_tmpa_box
```

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

```
1478 \box_clear_new:N \g_@@_line_box
```

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

```
1479 \vbox_set:Nn \l_tmpa_box
1480 {
1481   \vbox_unpack_drop:N \g_@@_output_box
1482   \bool_gset_false:N \g_tmpa_bool
1483   \unskip \unskip
1484   \bool_gset_false:N \g_tmpa_bool
1485   \bool_do_until:nn \g_tmpa_bool
1486   {
1487     \unskip \unskip \unskip
1488     \unpenalty \unkern
1489     \box_set_to_last:N \l_@@_line_box
1490     \box_if_empty:NTF \l_@@_line_box
1491     { \bool_gset_true:N \g_tmpa_bool }
1492     {
1493       \vbox_gset:Nn \g_tmpa_box
1494       {
1495         \vbox_unpack:N \g_tmpa_box
1496         \box_use:N \l_@@_line_box
1497       }
1498     }
1499   }
1500 }
```

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

```
1501 \bool_gset_false:N \g_tmpa_bool
1502 \int_zero:N \g_@@_line_int
1503 \bool_do_until:nn \g_tmpa_bool
1504 {
```

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`.

```
1505 \vbox_gset:Nn \g_tmpa_box
1506 {
1507   \vbox_unpack_drop:N \g_tmpa_box
1508   \box_gset_to_last:N \g_@@_line_box
1509 }
```

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.

```
1510 \box_if_empty:NTF \g_@@_line_box
1511 { \bool_gset_true:N \g_tmpa_bool }
1512 {
1513   \box_use:N \g_@@_line_box
1514   \int_gincr:N \g_@@_line_int
1515   \par
1516   \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`.

```
1517 \@@_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.

```
1518 \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int }
1519 { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int } }
1520 \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1521 { \mode_leave_vertical: }
1522 }
1523 }
1524 \skip_vertical:n { 2.5 pt }
1525 }
```

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

```

1526 \cs_new_protected:Npn \@@_composition_ii:
1527 {
1528     \use:e { \begin { minipage } [ \l_@@_box_str ] }
1529     { \l_@@_listing_width_dim }

```

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

```

1530     \vbox_unpack:N \g_@@_output_box

```

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

```

1531     \kern 2.5 pt
1532     \end { minipage }
1533 }

```

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

```

1534 \cs_new_protected:Npn \@@_composition_iii:
1535 {
1536     \use:e
1537     {
1538         \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.

```

1539         [ breakable , text~width = \l_@@_listing_width_dim ]
1540     }
1541     \par
1542     \vbox_unpack:N \g_@@_output_box
1543     \end { tcolorbox }
1544 }

```

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

```

1545 \cs_new_protected:Npn \@@_composition_iv:
1546 {
1547     \use:e
1548     {
1549         \begin { tcolorbox }
1550         [
1551             hbox ,
1552             text~width = \l_@@_listing_width_dim ,
1553             nobeforeafter ,
1554             box~align =
1555                 \str_case:Nn \l_@@_box_str
1556                 {
1557                     t { top }
1558                     b { bottom }
1559                     c { center }
1560                     m { center }
1561                 }
1562         ]
1563     }
1564     \box_use:N \g_@@_output_box
1565     \end { tcolorbox }
1566 }

```

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.

```

1567 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1568 {
1569     \int_case:nn
1570     {
1571         \lua_now:e
1572         {

```

```

1573         tex.sprint
1574         ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1575     }
1576 }
1577 { 1 { \penalty 100 } 2 \nobreak }
1578 }

```

`\@@_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`.

```

1579 \cs_new_protected:Npn \@@_create_output_box:
1580 {
1581     \@@_compute_code_width:
1582     \vbox_gset:Nn \g_@@_output_box
1583     { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1584     \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1585     \bool_lazy_any:nT
1586     {
1587         { \int_compare_p:nNn \l_@@_bg_colors_int > \c_zero_int }
1588         { \g_@@_rowcolor_inside_bool }
1589         {
1590             \l_@@_line_numbers_bool
1591             &&
1592             \str_if_eq_p:ee \l_@@_line_numbers_position_str { right }
1593         }
1594     }
1595     \@@_add_bg_and_right_nb_to_output_box:
1596 }

```

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:`.

```

1597 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_output_box:
1598 {
1599     \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`.

```

1600     \vbox_set:Nn \l_tmpa_box
1601     {
1602         \vbox_unpack_drop:N \g_@@_output_box

```

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

```

1603     \bool_gset_false:N \g_tmpa_bool
1604     \unskip \unskip

```

We begin the loop.

```

1605     \bool_do_until:nn \g_tmpa_bool
1606     {
1607         \unskip \unskip \unskip
1608         \int_set_eq:NN \l_tmpa_int \lastpenalty
1609         \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...

```

1610         \box_set_to_last:N \l_@@_line_box
1611         \box_if_empty:NTF \l_@@_line_box
1612         { \bool_gset_true:N \g_tmpa_bool }
1613         {

```

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

```

1614         \vbox_gset:Nn \g_@@_output_box
1615         {

```

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`.

```

1616             \@@_add_bg_and_right_nb_to_line_and_use:
1617             \kern -2.5 pt
1618             \penalty \l_tmpa_int
1619             \vbox_unpack:N \g_@@_output_box
1620         }
1621     }
1622     \int_gdecr:N \g_@@_line_int
1623 }
1624 }
1625 }
```

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

```

1626 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1627 {
1628     \lua_now:e
1629     {
1630         piton.GobbleParseNoPrint
1631         (
1632             '\l_piton_language_str' ,
1633             \int_use:N \l_@@_gobble_int ,
1634             token.scan_argument ( )
1635         )
1636     }
1637 }
1638 \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.

```

1639 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1640 {
1641     \lua_now:e
1642     {
1643         piton.RetrieveGobbleParse
1644         (
1645             '\l_piton_language_str' ,
1646             \int_use:N \l_@@_gobble_int ,
1647             \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1648             { \int_eval:n { - \l_@@_splittable_int } }
1649             { \int_use:N \l_@@_splittable_int } ,
1650             token.scan_argument ( )
1651         )
1652     }
1653 }
1654 \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.

```

1655 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1656 {
1657     \lua_now:e
1658     {
1659         piton.RetrieveGobbleSplitParse
1660         (
1661             '\l_piton_language_str' ,
1662             \int_use:N \l_@@_gobble_int ,
1663             \int_use:N \l_@@_splittable_int ,
1664             token.scan_argument ( )
```

```

1665     )
1666   }
1667 }
1668 \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`.

```

1669 \bool_if:NTF \g_@@_beamer_bool
1670 {
1671   \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1672   {
1673     \keys_set:nn { PitonOptions } { #2 }
1674     \begin { actionenv } < #1 >
1675   }
1676   { \end { actionenv } }
1677 }
1678 {
1679   \NewPitonEnvironment { Piton } { 0 { } }
1680   { \keys_set:nn { PitonOptions } { #1 } }
1681   { }
1682 }

1683 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1684 {
1685   \mode_if_vertical:F { \par }
1686   \group_begin:
1687   \seq_concat:NNN
1688     \l_file_search_path_seq
1689     \l_@@_path_seq
1690     \l_file_search_path_seq
1691   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1692   {
1693     \@@_input_file:nn { #1 } { #2 }
1694     #4
1695   }
1696   { #5 }
1697   \group_end:
1698 }

1699 \cs_new_protected:Npn \@@_unknown_file:n #1
1700 { \msg_error:nnn { piton } { Unknown-file } { #1 } }

1701 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1702 {
1703   \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1704   {

```

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

```

1705     \iow_log:n { No-file-#3 }
1706     \@@_unknown_file:n { #3 }
1707   }
1708 }
1709 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1710 {
1711   \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1712   {

```

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

```

1713     \iow_log:n { No-file-#3 }
1714     \@@_unknown_file:n { #3 }
1715   }
1716 }
1717 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1718 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```
1719 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1720 {
```

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 `>`).

```
1721 \tl_if_novalue:nF { #1 }
1722 {
1723     \bool_if:NTF \g_@@_beamer_bool
1724     { \begin { uncoverenv } < #1 > }
1725     { \@@_error_or_warning:n { overlay~without~beamer } }
1726 }
1727 \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.

```
1728 \iow_log:e { (\l_@@_file_name_str) }
1729 \int_zero_new:N \l_@@_first_line_int
1730 \int_zero_new:N \l_@@_last_line_int
1731 \int_set_eq:NN \l_@@_last_line_int \c_max_int
1732 \bool_set_true:N \l_@@_in_PitonInputFile_bool
1733 \keys_set:nn { PitonOptions } { #2 }
1734 \bool_if:NT \l_@@_line_numbers_absolute_bool
1735 { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1736 \bool_if:nTF
1737 {
1738     (
1739         \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1740         || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1741     )
1742     && ! \str_if_empty_p:N \l_@@_begin_range_str
1743 }
1744 {
1745     \@@_error_or_warning:n { bad~range~specification }
1746     \int_zero:N \l_@@_first_line_int
1747     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1748 }
1749 {
1750     \str_if_empty:NF \l_@@_begin_range_str
1751     {
1752         \@@_compute_range:
1753         \bool_lazy_or:nnT
1754             \l_@@_marker_include_lines_bool
1755             { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1756         {
1757             \int_decr:N \l_@@_first_line_int
1758             \int_incr:N \l_@@_last_line_int
1759         }
1760     }
1761 }
1762 \@@_pre_composition:
1763 \bool_if:NT \l_@@_line_numbers_absolute_bool
1764 { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1765 \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1766 {
1767     \int_gset:Nn \g_@@_visual_line_int
1768     { \l_@@_number_lines_start_int - 1 }
1769 }
```

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

```
1770 \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1771 { \int_gzero:N \g_@@_visual_line_int }
1772 \lua_now:e
1773 {
```

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

```

1774         piton.ReadFile(
1775             '\l_@@_file_name_str' ,
1776             \int_use:N \l_@@_first_line_int ,
1777             \int_use:N \l_@@_last_line_int )
1778     }
1779     \@@_composition:
1780     \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.

```

1781     \tl_if_novalue:nF { #1 }
1782     { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1783 }

```

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

```

1784 \cs_new_protected:Npn \@@_compute_range:
1785 {

```

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

```

1786     \str_set:Nx \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1787     \str_set:Nx \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`.

```

1788     \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1789     \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1790     \lua_now:e
1791     {
1792         piton.ComputeRange
1793         ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1794     }
1795 }

```

2.8 The styles

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

```

1796 \NewDocumentCommand { \PitonStyle } { m }
1797 {
1798     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1799     { \use:c { pitonStyle _ #1 } }
1800 }

```

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`”.

```

1801 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1802 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1803 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1804 {
1805     \str_clear_new:N \l_@@_SetPitonStyle_option_str
1806     \str_set:Nx \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1807     \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1808     { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1809     \keys_set:nn { piton / Styles } { #2 }
1810 }

1811 \cs_new_protected:Npn \@@_math_scantokens:n #1
1812 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

```

```

1813 \clist_new:N \g_@@_styles_clist
1814 \clist_gset:Nn \g_@@_styles_clist
1815 {
1816     Comment ,
1817     Comment.Internal ,
1818     Comment.LaTeX ,
1819     Discard ,
1820     Delim ,
1821     Exception ,
1822     FormattingType ,
1823     Identifier.Internal ,
1824     Identifier ,
1825     InitialValues ,
1826     Interpol.Inside ,
1827     Keyword ,
1828     Keyword.Governing ,
1829     Keyword.Constant ,
1830     Keyword2 ,
1831     Keyword3 ,
1832     Keyword4 ,
1833     Keyword5 ,
1834     Keyword6 ,
1835     Keyword7 ,
1836     Keyword8 ,
1837     Keyword9 ,
1838     Name.Builtin ,
1839     Name.Class ,
1840     Name.Constructor ,
1841     Name.Decorator ,
1842     Name.Field ,
1843     Name.Function ,
1844     Name.Module ,
1845     Name.Namespace ,
1846     Name.Table ,
1847     Name.Type ,
1848     Number ,
1849     Number.Internal ,
1850     Operator ,
1851     Operator.Word ,
1852     Preproc ,
1853     Prompt ,
1854     Punct ,
1855     String.Doc ,
1856     String.Doc.Internal ,
1857     String.Interpol ,
1858     String.Long ,
1859     String.Long.Internal ,
1860     String.Short ,
1861     String.Short.Internal ,
1862     Tag ,
1863     TypeParameter ,
1864     UserFunction ,

```

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

```

1865     TypeExpression ,

```

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

```

1866     Directive
1867 }
1868 \clist_map_inline:Nn \g_@@_styles_clist
1869 {
1870     \keys_define:nn { piton / Styles }
1871     {
1872         #1 .value_required:n = true ,

```

```

1873     #1 .code:n =
1874     \tl_set:cn
1875     {
1876         pitonStyle _
1877         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1878         { \l_@@_SetPitonStyle_option_str _ }
1879         #1
1880     }
1881     { ##1 }
1882 }
1883 }
1884
1885 \keys_define:nn { piton / Styles }
1886 {
1887     String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1888     String      .value_required:n = true ,
1889     Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1890     Comment.Math .value_required:n = true ,
1891     unknown     .code:n = \@@_unknown_style:
1892 }

```

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.

```

1893 \cs_new_protected:Npn \@@_unknown_style:
1894 {
1895     \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1896     {
1897         \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1898         \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`.

```

1899     \bool_lazy_and:nnTF
1900     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1901     {
1902         \str_if_eq_p:Vn \l_tmpa_str { Module }
1903         ||
1904         \str_if_eq_p:Vn \l_tmpa_str { Type }
1905     }

```

Now, we will create a new style.

```

1906     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1907     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1908 }
1909 { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1910 }

```

```

1911 \SetPitonStyle[OCaml]
1912 {
1913     TypeExpression =
1914     {
1915         \SetPitonStyle [ OCaml ]
1916         {
1917             Identifier = \PitonStyle { Name.Type } ,
1918             Name.Builtin = \PitonStyle { Name.Type }
1919         }
1920         \@@_piton:n
1921     }
1922 }

```

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`.

```

1923 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that clist.

```

1924 \clist_gsort:Nn \g_@@_styles_clist
1925 {
1926   \str_compare:nNnTF { #1 } < { #2 }
1927     \sort_return_same:
1928     \sort_return_swapped:
1929 }

1930 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1931
1932 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1933
1934 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1935 {
1936   \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`.

```

1937   \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space
1938   \seq_clear:N \l_tmpa_seq
1939   \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1940   \seq_use:Nn \l_tmpa_seq { \- }
1941 }

```

```

1942 \cs_new_protected:Npn \@@_comment:n #1
1943 { \PitonStyle { Comment } { \PitonSpaceSubstitute { #1 } } }

```

We use a standard name for the following command (and not a 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”.

```

1944 \cs_new_protected:Npn \PitonSpaceSubstitute #1 % noqa
1945 {
1946   \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1947   {
1948     \tl_set:Nn \l_tmpa_tl { #1 }
1949     \tl_replace_all:NVn \l_tmpa_tl
1950       \c_catcode_other_space_tl
1951       \@@_breakable_space:
1952     \l_tmpa_tl
1953   }
1954   { #1 }
1955 }

```

```

1956 \cs_new_protected:Npn \@@_string_long:n #1
1957 {
1958   \PitonStyle { String.Long }
1959   {
1960     \bool_if:NTF \l_@@_break_strings_anywhere_bool
1961     { \@@_actually_break_anywhere:n { #1 } }
1962     {

```

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.

```

1963       \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1964       {
1965         \tl_set:Nn \l_tmpa_tl { #1 }

```

```

1966         \tl_replace_all:NVn \l_tmpa_tl
1967         \c_catcode_other_space_tl
1968         \@@_breakable_space:
1969         \l_tmpa_tl
1970     }
1971     { #1 }
1972 }
1973 }
1974 }
1975 \cs_new_protected:Npn \@@_string_short:n #1
1976 {
1977     \PitonStyle { String.Short }
1978     {
1979         \bool_if:NT \l_@@_break_strings_anywhere_bool
1980         { \@@_actually_break_anywhere:n }
1981         { #1 }
1982     }
1983 }
1984 \cs_new_protected:Npn \@@_string_doc:n #1
1985 {
1986     \PitonStyle { String.Doc }
1987     {
1988         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1989         {
1990             \tl_set:Nn \l_tmpa_tl { #1 }
1991             \tl_replace_all:NVn \l_tmpa_tl
1992             \c_catcode_other_space_tl
1993             \@@_breakable_space:
1994             \l_tmpa_tl
1995         }
1996         { #1 }
1997     }
1998 }
1999 \cs_new_protected:Npn \@@_number:n #1
2000 {
2001     \PitonStyle { Number }
2002     {
2003         \bool_if:NT \l_@@_break_numbers_anywhere_bool
2004         { \@@_actually_break_anywhere:n }
2005         { #1 }
2006     }
2007 }

```

2.9 The initial styles

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

```

2008 \SetPitonStyle
2009 {
2010     Comment           = \color [ HTML ] { 0099FF } \itshape ,
2011     Comment.Internal  = \@@_comment:n ,
2012     Exception         = \color [ HTML ] { CC0000 } ,
2013     Keyword           = \color [ HTML ] { 006699 } \bfseries ,
2014     Keyword.Governing = \color [ HTML ] { 006699 } \bfseries ,
2015     Keyword.Constant  = \color [ HTML ] { 006699 } \bfseries ,
2016     Name.Builtin      = \color [ HTML ] { 336666 } ,
2017     Name.Decorator     = \color [ HTML ] { 9999FF } ,
2018     Name.Class        = \color [ HTML ] { 00AA88 } \bfseries ,
2019     Name.Function      = \color [ HTML ] { CC00FF } ,
2020     Name.Namespace    = \color [ HTML ] { 00CCFF } ,
2021     Name.Constructor  = \color [ HTML ] { 006000 } \bfseries ,
2022     Name.Field        = \color [ HTML ] { AA6600 } ,

```

```

2023 Name.Module          = \color [ HTML ] { 0060A0 } \bfseries ,
2024 Name.Table           = \color [ HTML ] { 309030 } ,
2025 Number               = \color [ HTML ] { FF6600 } ,
2026 Number.Internal      = \@@_number:n ,
2027 Operator             = \color [ HTML ] { 555555 } ,
2028 Operator.Word        = \bfseries ,
2029 String               = \color [ HTML ] { CC3300 } ,
2030 String.Long.Internal = \@@_string_long:n ,
2031 String.Short.Internal = \@@_string_short:n ,
2032 String.Doc.Internal  = \@@_string_doc:n ,
2033 String.Doc           = \color [ HTML ] { CC3300 } \itshape ,
2034 String.Interpol      = \color [ HTML ] { AA0000 } ,
2035 Comment.LaTeX        = \normalfont \color [ rgb ] { .468, .532, .6 } ,
2036 Name.Type            = \color [ HTML ] { 336666 } ,
2037 InitialValues       = \@@_piton:n ,
2038 Interpol.Inside      = { \l_@@_font_command_tl \@@_piton:n } ,
2039 TypeParameter        = \color [ HTML ] { 336666 } \itshape ,
2040 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).

```

2041 Identifier.Internal  = \@@_identifier:n ,
2042 Identifier            = ,
2043 Directive            = \color [ HTML ] { AA6600 } ,
2044 Tag                  = \colorbox { gray!10 } ,
2045 UserFunction         = \PitonStyle { Identifier } ,
2046 Prompt              = ,
2047 Discard              = \use_none:n
2048 }

```

2.10 Styles specific to the language expl

```

2049 \clist_new:N \g_@@_expl_styles_clist
2050 \clist_gset:Nn \g_@@_expl_styles_clist
2051 {
2052   Scope.l ,
2053   Scope.g ,
2054   Scope.c
2055 }
2056 \clist_map_inline:Nn \g_@@_expl_styles_clist
2057 {
2058   \keys_define:nn { piton / Styles }
2059   {
2060     #1 .value_required:n = true ,
2061     #1 .code:n =
2062       \tl_set:cn
2063       {
2064         pitonStyle _
2065         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
2066         { \l_@@_SetPitonStyle_option_str _ }
2067         #1
2068       }
2069     { ##1 }
2070   }
2071 }
2072 \SetPitonStyle [ expl ]
2073 {
2074   Scope.l      = ,
2075   Scope.g      = \bfseries ,
2076   Scope.c      = \slshape ,

```

```

2077 Type.bool      = \color [ HTML ] { AA6600 } ,
2078 Type.box       = \color [ HTML ] { 267910 } ,
2079 Type.clist     = \color [ HTML ] { 309030 } ,
2080 Type.fp        = \color [ HTML ] { FF3300 } ,
2081 Type.int       = \color [ HTML ] { FF6600 } ,
2082 Type.seq       = \color [ HTML ] { 309030 } ,
2083 Type.skip      = \color [ HTML ] { 0CC060 } ,
2084 Type.str       = \color [ HTML ] { CC3300 } ,
2085 Type.tl        = \color [ HTML ] { AA2200 } ,
2086 Module.bool    = \color [ HTML ] { AA6600 } ,
2087 Module.box     = \color [ HTML ] { 267910 } ,
2088 Module.cs      = \bfseries \color [ HTML ] { 006699 } ,
2089 Module.exp     = \bfseries \color [ HTML ] { 404040 } ,
2090 Module.hbox    = \color [ HTML ] { 267910 } ,
2091 Module.prg     = \bfseries ,
2092 Module.clist   = \color [ HTML ] { 309030 } ,
2093 Module.fp      = \color [ HTML ] { FF3300 } ,
2094 Module.int     = \color [ HTML ] { FF6600 } ,
2095 Module.seq     = \color [ HTML ] { 309030 } ,
2096 Module.skip    = \color [ HTML ] { 0CC060 } ,
2097 Module.str     = \color [ HTML ] { CC3300 } ,
2098 Module.tl      = \color [ HTML ] { AA2200 } ,
2099 Module.vbox    = \color [ HTML ] { 267910 }
2100 }

```

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).

```

2101 \hook_gput_code:nnn { begindocument } { . }
2102 {
2103   \bool_if:NT \g_@@_math_comments_bool
2104     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
2105 }

```

2.11 Highlighting some identifiers

```

2106 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
2107 {
2108   \clist_set:Nn \l_tmpa_clist { #2 }
2109   \tl_if_novalue:nTF { #1 }
2110   {
2111     \clist_map_inline:Nn \l_tmpa_clist
2112       { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
2113   }
2114   {
2115     \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
2116     \str_if_eq:onT \l_tmpa_str { current-language }
2117       { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
2118     \clist_map_inline:Nn \l_tmpa_clist
2119       { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
2120   }
2121 }
2122 \cs_new_protected:Npn \@@_identifier:n #1
2123 {
2124   \str_set:Nn \l_tmpa_str { #1 }
2125   \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2126   {
2127     \cs_if_exist_use:cF { PitonIdentifier _ \l_tmpa_str }
2128     { \PitonStyle { Identifier } }
2129   }
2130   { #1 }
2131 }

```

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`).

```
2132 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
2133 {
```

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

```
2134 { \PitonStyle { Name.Function } { #1 } }
2135 \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}`.

```
2136 \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2137 { \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`.**

```
2138 \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
2139 { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
2140 \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.

```
2141 \seq_if_in:Nof { g_@@_languages_seq } { \l_piton_language_str }
2142 { \seq_gput_left:No { g_@@_languages_seq } { \l_piton_language_str } }
2143 }
```

```
2144 \NewDocumentCommand \PitonClearUserFunctions { ! o }
2145 {
2146 \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.

```
2147 { \@@_clear_all_functions: }
2148 { \@@_clear_list_functions:n { #1 } }
2149 }
```

```
2150 \cs_new_protected:Npn \@@_clear_list_functions:n #1
2151 {
2152 \clist_set:Nn \l_tmpa_clist { #1 }
2153 \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
2154 \clist_map_inline:nn { #1 }
2155 { \seq_gremove_all:Nn { g_@@_languages_seq } { ##1 } }
2156 }
```

```
2157 \cs_new_protected:Npn \@@_clear_functions_i:n #1
2158 { \@@_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).

```
2159 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2160 {
2161 \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2162 {
2163 \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2164 { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }
2165 \seq_gclear:c { g_@@_functions _ #1 _ seq }
2166 }
2167 }
2168 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }
```

```
2169 \cs_new_protected:Npn \@@_clear_functions:n #1
```

```

2170 {
2171   \@@_clear_functions_i:n { #1 }
2172   \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2173 }

```

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

```

2174 \cs_new_protected:Npn \@@_clear_all_functions:
2175 {
2176   \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2177   \seq_gclear:N \g_@@_languages_seq
2178 }

```

```

2179 \AtEndDocument
2180 {

```

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

```

2181   \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.

```

2182   \IfPDFManagementActiveTF
2183     { \@@_join_files: }
2184     { \@@_join_files_legacy: }
2185 }

```

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

```

2186 \cs_new_protected:Npn \@@_join_files:
2187 {
2188   \seq_map_inline:Nn \g_@@_join_seq
2189   {
2190     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2191     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2192     \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2193     {
2194       <<
2195         /Type /Filespec
2196         /UF <\l_tmpa_str>
2197         /EF << /F~\pdf_object_ref_last: >>
2198         /Desc (Computer~listing)
2199         /AFRelationship /Supplement
2200       >>
2201     }
2202   }
2203 }

```

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

```

2204 \cs_new_protected:Npn \@@_join_files_legacy:
2205 {
2206   \seq_map_inline:Nn \g_@@_join_seq
2207   {
2208     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2209     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2210     \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`.

```

2211     {
2212       /Subtype /FileAttachment

```

```

2213         /F~2
2214         /Name /Paperclip
2215         /Contents (Computer~listing)
2216         /FS <<
2217             /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).

```

2218         /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`).

```

2219         /EF << /F~\pdf_object_ref_last: >>
2220         /AFRelationship /Supplement
2221     >>
2222 }
2223 }
2224 }

```

2.12 Spaces of indentation

```

2225 \cs_new_protected:Npn \@@_define_leading_space_normal:
2226 {
2227     \cs_set_protected:Npn \@@_leading_space:
2228     {
2229         \int_gincr:N \g_@@_indentation_int

```

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

```

2230         \hbox:n { ~ }
2231     }
2232 }
2233 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2234 {
2235     \cs_set_protected:Npn \@@_leading_space:
2236     {
2237         \int_gincr:N \g_@@_indentation_int
2238         \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2239         {
2240             \color { white }
2241             \transparent { 0 }
2242             . % previously : □ U+2423
2243         }
2244         \pdfextension literal { EMC }
2245     }
2246 }
2247 \@@_define_leading_space_Foxit:

```

2.13 Security

```

2248 \AddToHook { env / piton / before }
2249 { \@@_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.

```

2250 \cs_new_protected:Npn \@@_unknown_key:nn #1 #2
2251 {

```

```

2252 \str_set_eq:NN \l_tmpa_str \l_keys_key_str
2253 \str_replace_all:Nnn \l_tmpa_str { ~ } { - }
2254 \str_set:Nx \l_tmpa_str { \str_lowercase:f { \l_tmpa_str } }
2255 \bool_set_false:N \l_tmpa_bool
2256 \clist_map_inline:nn { #1 }
2257 {
2258     \keys_if_exist:neT { ##1 } { \l_tmpa_str }
2259     {
2260         \@@_error:n { key~with~normal~form~exists }
2261         \bool_set_true:N \l_tmpa_bool
2262         \clist_map_break:
2263     }
2264 }
2265 \bool_if:NF \l_tmpa_bool { \@@_error:n { #2 } }
2266 }

2267 \@@_msg_new:nn { key~with~normal~form~exists }
2268 {
2269     The~key~'\l_keys_key_str'~does~not~exists.~It~will~be~ignored.\\
2270     Maybe~you~want~to~use~the~key~'\l_tmpa_str'.
2271 }

2272 \@@_msg_new:nn { No~environment~piton }
2273 {
2274     There~is~no~environment~piton!\\
2275     There~is~an~environment~{Piton}~and~a~command~
2276     \token_to_str:N \piton\ but~there~is~no~environment~
2277     {piton}.
2278 }

2279 \@@_msg_new:nn { rounded~corners~without~Tikz }
2280 {
2281     TikZ~not~used \\
2282     You~can't~use~the~key~'rounded~corners'~because~
2283     you~have~not~loaded~the~package~TikZ.~
2284     If~you~go~on,~your~key~will~be~ignored.~
2285     You~won't~have~similar~error~till~the~end~of~the~document.
2286 }

2287 \@@_msg_new:nn { tcolorbox~not~loaded }
2288 {
2289     tcolorbox~not~loaded \\
2290     You~can't~use~the~key~'tcolorbox'~because~
2291     you~have~not~loaded~the~package~tcolorbox.~
2292     Use~\token_to_str:N \usepackage[breakable]{tcolorbox}.~
2293     If~you~go~on,~that~key~will~be~ignored.
2294 }

2295 \@@_msg_new:nn { library~breakable~not~loaded }
2296 {
2297     breakable~not~loaded \\
2298     You~can't~use~the~key~'tcolorbox'~because~
2299     you~have~not~loaded~the~library~'breakable'~of~tcolorbox'.~
2300     You~should~load~'piton'~with~the~key~'breakable'~or~
2301     use~\token_to_str:N \tcbuselibrary{breakable}~
2302     in~the~preamble~of~your~document.~
2303     If~you~go~on,~that~key~will~be~ignored.
2304 }

2305 \@@_msg_new:nn { Language~not~defined }
2306 {
2307     Language~not~defined \\
2308     The~language~'\l_tmpa_tl'~has~not~been~defined~previously.~
2309     If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2310     will~be~ignored.
2311 }

2312 \@@_msg_new:nn { bad~version~of~piton.lua }

```

```

2313 {
2314   Bad~number~version~of~'piton.lua'\
2315   The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2316   version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2317   address~that~issue.
2318 }
2319 \@@_msg_new:nn { Unknown~key~NewPitonLanguage }
2320 {
2321   Unknown~key~for~\token_to_str:N \NewPitonLanguage.\
2322   The~key~'\l_keys_key_str'~is~unknown.~
2323   This~key~will~be~ignored.
2324 }
2325 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
2326 {
2327   The~style~'\l_keys_key_str'~is~unknown.\
2328   This~setting~will~be~ignored.~
2329   The~available~styles~are~(in~alphabetic~order):~
2330   \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2331 }
2332 \@@_msg_new:nn { Invalid~key }
2333 {
2334   Wrong~use~of~key.\
2335   You~can't~use~the~key~'\l_keys_key_str'~here.~
2336   Your~key~will~be~ignored.
2337 }
2338 \@@_msg_new:nn { Unknown~key~for~line~numbers }
2339 {
2340   Unknown~key. \
2341   The~key~'line~numbers / \l_keys_key_str'~is~unknown.~
2342   The~available~keys~of~the~family~'line~numbers'~are~(in~
2343   alphabetic~order):~
2344   absolute,~false,~format,~label~empty~lines,~lmonono10~drawn,~
2345   ~position,~resume,~skip~empty~lines,~sep,~start~and~true.~
2346   Your~key~will~be~ignored.
2347 }
2348 \@@_msg_new:nn { Unknown~key~for~marker }
2349 {
2350   Unknown~key. \
2351   The~key~'marker / \l_keys_key_str'~is~unknown.~
2352   The~available~keys~of~the~family~'marker'~are~(in~
2353   alphabetic~order):~ beginning,~end~and~include~lines.~
2354   Your~key~will~be~ignored.
2355 }
2356 \@@_msg_new:nn { bad~range~specification }
2357 {
2358   Incompatible~keys.\
2359   You~can't~specify~the~range~of~lines~to~include~by~using~both~
2360   markers~and~explicit~number~of~lines.~
2361   Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2362 }
2363 \cs_new_nopar:Nn \@@_thepage:
2364 {
2365   \thepage
2366   \cs_if_exist:NT \insertframenumber
2367   {
2368     ~(frame~\insertframenumber
2369     \cs_if_exist:NT \beamer@slidenum { ,~slide~\insertslidenum }
2370     )
2371   }
2372 }

```

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`.

```

2373 \@@_msg_new:nn { SyntaxError }
2374 {
2375   Syntax-Error-on-page-\@@_thepage:.\
2376   Your-code-of-the-language-\l_piton_language_str'-is-not~
2377   syntactically~correct.~
2378   It~won't~be~printed-in~the~PDF~file.
2379 }
2380 \@@_msg_new:nn { FileError }
2381 {
2382   File-Error.\
2383   It's-not-possible-to-write-on-the-file-#1' \
2384   \sys_if_shell_unrestricted:F
2385   { (try-to-compile-with-'lualatex--shell-escape').\ }
2386   If-you-go-on,~nothing-will-be-written-on-that-file.
2387 }
2388 \@@_msg_new:nn { InexistentDirectory }
2389 {
2390   Inexistent-directory.\
2391   The-directory-\l_@@_path_write_str'~
2392   given-in-the-key-'path-write'~does-not-exist.~
2393   Nothing-will-be-written-on-\l_@@_write_str'.
2394 }
2395 \@@_msg_new:nn { begin-marker-not-found }
2396 {
2397   Marker-not-found.\
2398   The-range-\l_@@_begin_range_str'~provided-to-the~
2399   command~\token_to_str:N \PitonInputFile\ has-not-been-found.~
2400   The-whole-file-\l_@@_file_name_str'~will-be-inserted.
2401 }
2402 \@@_msg_new:nn { end-marker-not-found }
2403 {
2404   Marker-not-found.\
2405   The-marker-of-end-of-the-range-\l_@@_end_range_str'~
2406   provided-to-the-command~\token_to_str:N \PitonInputFile\
2407   has-not-been-found.~The-file-\l_@@_file_name_str'~will~
2408   be-inserted~till~the~end.
2409 }
2410 \@@_msg_new:nn { Unknown-file }
2411 {
2412   Unknown-file. \
2413   The-file-#1'~is-unknown.~
2414   Your-command~\token_to_str:N \PitonInputFile\ will-be-ignored.
2415 }
2416 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2417 {
2418   \bool_if:NF \g_@@_beamer_bool
2419   { \@@_error_or_warning:n { Without~beamer } }
2420 }
2421 \@@_msg_new:nn { Without~beamer }
2422 {
2423   Key-\l_keys_key_str'~without~Beamer.\
2424   You-should-not-use-the-key-\l_keys_key_str'~since~you~
2425   are-not-in-Beamer.~However,~you~can~go~on.
2426 }
2427 \@@_msg_new:nn { rowcolor~in-detected-commands }
2428 {
2429   'rowcolor'~forbidden-in-'detected-commands'.\

```

```

2430     You-should-put~'rowcolor'~in~'raw-detected-commands'.~
2431     That-key~will~be~ignored.
2432 }
2433 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2434 {
2435     Unknown~key. \\
2436     The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
2437     It~will~be~ignored.~
2438     For~a~list~of~the~available~keys,~type~H~<return>.
2439 }
2440 {
2441     The~available~keys~are~(in~alphabetic~order):~
2442     annotation,~
2443     add-to-split-separation,~
2444     after-before-escape,~
2445     auto-gobble,~
2446     background-color,~
2447     before-end-escape,~
2448     begin-range,~
2449     box,~
2450     break-lines,~
2451     break-lines-in-piton,~
2452     break-lines-in-Piton,~
2453     break-numbers-anywhere,~
2454     break-strings-anywhere,~
2455     continuation-symbol,~
2456     continuation-symbol-on-indentation,~
2457     detected-beamer-commands,~
2458     detected-beamer-environments,~
2459     detected-commands,~
2460     end-of-broken-line,~
2461     end-range,~
2462     env-gobble,~
2463     env-used-by-split,~
2464     font-command(+),~
2465     gobble,~
2466     indent-broken-lines,~
2467     join,~
2468     label-as-zlabel,~
2469     language,~
2470     left-margin,~
2471     line-numbers/,~
2472     marker/,~
2473     math-comments,~
2474     no-join,~
2475     no-write,~
2476     path,~
2477     path-write,~
2478     print,~
2479     prompt-background-color,~
2480     raw-detected-commands,~
2481     resume,~
2482     right-margin,~
2483     rounded-corners,~
2484     show-spaces,~
2485     show-spaces-in-strings,~
2486     splittable,~
2487     splittable-on-empty-lines,~
2488     split-on-empty-lines,~
2489     split-separation,~
2490     tabs-auto-gobble,~
2491     tab-size,~
2492     tcolorbox,~

```

```

2493     varwidth,~
2494     vertical-detected-commands,~
2495     width-and-write.
2496 }

2497 \@@_msg_new:nn { label-with-lines-numbers }
2498 {
2499     You~can't~use~the~command~\token_to_str:N \label\
2500     or~\token_to_str:N \zlabel\
2501     because~the~key~'line-numbers'~
2502     is~not~active.~If~you~go~on,~that~command~will~ignored.
2503 }

2504 \@@_msg_new:nn { overlay-without-beamer }
2505 {
2506     You~can't~use~an~argument~<...>~for~your~command~
2507     \token_to_str:N \PitonInputFile\ because~you~are~not~
2508     in~Beamer.~If~you~go~on,~that~argument~will~be~ignored.
2509 }

2510 \@@_msg_new:nn { label-as-zlabel-needs-zref-package }
2511 {
2512     The~key~'label-as-zlabel'~requires~the~package~'zref'.~
2513     Please~load~the~package~'zref'~before~setting~the~key.
2514 }
2515 \hook_gput_code:nnn { begindocument } { . }
2516 {
2517     \bool_if:NT \g_@@_label_as_zlabel_bool
2518     {
2519         \IfPackageLoadedF { zref-base }
2520         { \@@_fatal:n { label-as-zlabel-needs-zref-package } }
2521     }
2522 }

```

2.15 The glyphs of the 10 arabic numbers

```

2523 \cs_new_protected:Npn \@@_draw_number:n #1
2524 { \tl_map_inline:nn { #1 } { \use:c { c_@@_glyph_ ##1 _tl } } }
2525 \cs_generate_variant:Nn \@@_draw_number:n { e }

2526 \cs_set_eq:NN \tlconstcn \tl_const:cn

2527 \ExplSyntaxOff
2528 \tlconstcn{c_@@_glyph_0_tl}
2529 {%
2530     \hbox to 5pt
2531     {%
2532         \hfill
2533         \pdfextension literal
2534         {
2535             4.24 3.05 m
2536             4.24 4.91 3.22 6.22 2.12 6.22 c
2537             1.00 6.22 0 4.88 0 3.06 c
2538             0 1.20 1.02 -0.11 2.12 -0.11 c
2539             3.24 -0.11 4.24 1.23 4.24 3.05 c
2540             3.55 3.16 m
2541             3.55 1.68 2.90 0.50 2.12 0.50 c
2542             1.34 0.50 0.69 1.68 0.69 3.16 c
2543             0.69 4.62 1.38 5.61 2.12 5.61 c
2544             2.85 5.61 3.55 4.63 3.55 3.16 c

```

```

2545         h f
2546     }%
2547 \hfill
2548 }%
2549 }
2550 \tlconstcn{c_@@_glyph_1_tl}
2551 {%
2552     \hbox to 5pt
2553     {%
2554         \hfill \kern 1 pt
2555         \pdfextension literal
2556         {
2557             3.37 0.3 m
2558             3.37 0.61 3.13 0.61 2.97 0.61 c
2559             2.06 0.61 l
2560             2.06 5.81 l
2561             2.06 5.97 2.06 6.22 1.76 6.22 c
2562             1.57 6.22 1.51 6.1 1.46 5.98 c
2563             1.08 5.13 0.56 5.02 0.37 5.0 c
2564             0.21 4.99 0.0 4.97 0.0 4.69 c
2565             0.0 4.44 0.18 4.39 0.33 4.39 c
2566             0.52 4.39 0.93 4.45 1.37 4.83 c
2567             1.37 0.61 l
2568             0.46 0.61 l
2569             0.3 0.61 0.06 0.61 0.06 0.3 c
2570             0.06 0.0 0.31 0.0 0.46 0.0 c
2571             2.97 0.0 l
2572             3.12 0.0 3.37 0.0 3.37 0.3 c
2573         h f
2574     }%
2575 \hfill
2576 }%
2577 }
2578 \tlconstcn{c_@@_glyph_2_tl}
2579 {%
2580     \hbox to 5pt
2581     {%
2582         \hfill
2583         \pdfextension literal
2584         {
2585             4.2 0.41 m
2586             4.2 0.67 l
2587             4.2 0.86 4.2 1.08 3.86 1.08 c
2588             3.51 1.08 3.51 0.89 3.51 0.61 c
2589             1.13 0.61 l
2590             1.72 1.12 2.68 1.87 3.11 2.27 c
2591             3.74 2.83 4.2 3.47 4.2 4.27 c
2592             4.2 5.47 3.19 6.22 1.97 6.22 c
2593             0.79 6.22 0.0 5.4 0.0 4.55 c
2594             0.0 4.18 0.28 4.07 0.45 4.07 c
2595             0.66 4.07 0.89 4.24 0.89 4.52 c
2596             0.89 4.64 0.84 4.77 0.75 4.84 c
2597             0.9 5.3 1.37 5.61 1.92 5.61 c
2598             2.74 5.61 3.51 5.15 3.51 4.27 c
2599             3.51 3.57 3.02 2.99 2.36 2.44 c
2600             0.15 0.58 l
2601             0.06 0.5 0.0 0.45 0.0 0.31 c
2602             0.0 0.0 0.25 0.0 0.41 0.0 c
2603             3.8 0.0 l
2604             4.13 0.0 4.2 0.09 4.2 0.41 c
2605         h f
2606     }%
2607 \hfill

```

```

2608 }%
2609 }
2610 \tlconstcn{c_@@_glyph_3_tl}
2611 {%
2612   \hbox to 5pt
2613   {%
2614     \hfill
2615     \pdfextension literal
2616     {
2617       4.36 1.74 m
2618       4.36 2.45 3.89 3.04 3.23 3.34 c
2619       3.79 3.7 4.07 4.27 4.07 4.81 c
2620       4.07 5.55 3.34 6.22 2.19 6.22 c
2621       0.99 6.22 0.29 5.74 0.29 5.05 c
2622       0.29 4.72 0.54 4.58 0.74 4.58 c
2623       0.95 4.58 1.18 4.75 1.18 5.03 c
2624       1.18 5.17 1.12 5.27 1.09 5.3 c
2625       1.4 5.61 2.11 5.61 2.2 5.61 c
2626       2.88 5.61 3.38 5.25 3.38 4.8 c
2627       3.38 4.5 3.23 4.15 2.96 3.93 c
2628       2.64 3.67 2.39 3.65 2.03 3.63 c
2629       1.46 3.59 1.31 3.59 1.31 3.3 c
2630       1.31 2.99 1.55 2.99 1.71 2.99 c
2631       2.17 2.99 l
2632       3.16 2.99 3.67 2.32 3.67 1.74 c
2633       3.67 1.13 3.11 0.5 2.2 0.5 c
2634       1.8 0.5 1.03 0.61 0.77 1.08 c
2635       0.82 1.13 0.89 1.19 0.89 1.39 c
2636       0.89 1.63 0.7 1.83 0.45 1.83 c
2637       0.22 1.83 0.0 1.68 0.0 1.36 c
2638       0.0 0.47 0.97 -0.11 2.2 -0.11 c
2639       3.51 -0.11 4.36 0.81 4.36 1.74 c
2640     h f
2641   }%
2642   \hfill
2643 }%
2644 }
2645 \tlconstcn{c_@@_glyph_4_tl}
2646 {%
2647   \hbox to 5pt
2648   {%
2649     \hfill
2650     \pdfextension literal
2651     {
2652       4.66 1.99 m
2653       4.66 2.3 4.42 2.3 4.26 2.3 c
2654       3.48 2.3 l
2655       3.48 5.82 l
2656       3.48 6.15 3.41 6.23 3.07 6.23 c
2657       2.79 6.23 l
2658       2.55 6.23 2.5 6.22 2.37 6.02 c
2659       0.09 2.44 l
2660       0.0 2.31 0.0 2.29 0.0 2.09 c
2661       0.0 1.76 0.09 1.69 0.4 1.69 c
2662       2.92 1.69 l
2663       2.92 0.61 l
2664       2.3 0.61 l
2665       2.14 0.61 1.9 0.61 1.9 0.3 c
2666       1.9 0.0 2.15 0.0 2.3 0.0 c
2667       4.1 0.0 l
2668       4.25 0.0 4.5 0.0 4.5 0.3 c
2669       4.5 0.61 4.26 0.61 4.1 0.61 c
2670       3.48 0.61 l

```

```

2671         3.48 1.69 l
2672         4.26 1.69 l
2673         4.41 1.69 4.66 1.69 4.66 1.99 c
2674         2.92 2.3 m
2675         0.7 2.3 l
2676         2.92 5.79 l
2677         h f
2678     }%
2679 \hfill
2680 }%
2681 }
2682 \tlconstcn{c_@@_glyph_5_tl}
2683 {%
2684     \hbox to 5pt
2685     {%
2686         \hfill
2687         \pdfextension literal
2688         {
2689             4.2 1.9 m
2690             4.2 2.91 3.43 3.89 2.25 3.89 c
2691             1.9 3.89 1.46 3.82 1.05 3.6 c
2692             1.05 5.5 l
2693             3.45 5.5 l
2694             3.6 5.5 3.85 5.5 3.85 5.8 c
2695             3.85 6.11 3.61 6.11 3.45 6.11 c
2696             0.76 6.11 l
2697             0.43 6.11 0.36 6.02 0.36 5.7 c
2698             0.36 3.04 l
2699             0.36 2.86 0.36 2.63 0.68 2.63 c
2700             0.86 2.63 0.9 2.68 0.98 2.78 c
2701             1.25 3.1 1.66 3.28 2.24 3.28 c
2702             3.06 3.28 3.51 2.55 3.51 1.9 c
2703             3.51 1.1 2.8 0.5 1.96 0.5 c
2704             1.68 0.5 1.04 0.58 0.76 1.13 c
2705             0.81 1.18 0.89 1.25 0.89 1.45 c
2706             0.89 1.74 0.66 1.9 0.45 1.9 c
2707             0.3 1.9 0.0 1.81 0.0 1.42 c
2708             0.0 0.59 0.84 -0.11 1.96 -0.11 c
2709             3.21 -0.11 4.2 0.79 4.2 1.9 c
2710             h f
2711         }%
2712     \hfill
2713 }%
2714 }
2715 \tlconstcn{c_@@_glyph_6_tl}
2716 {%
2717     \hbox to 5pt
2718     {%
2719         \hfill
2720         \pdfextension literal
2721         {
2722             4.18 1.93 m
2723             4.18 3.07 3.3 3.96 2.2 3.96 c
2724             1.68 3.96 1.16 3.79 0.71 3.38 c
2725             0.85 4.79 1.79 5.61 2.67 5.61 c
2726             3.01 5.61 3.17 5.5 3.22 5.45 c
2727             3.17 5.4 3.12 5.34 3.12 5.16 c
2728             3.12 4.92 3.3 4.72 3.56 4.72 c
2729             3.81 4.72 4.01 4.89 4.01 5.19 c
2730             4.01 5.68 3.65 6.22 2.68 6.22 c
2731             1.34 6.22 0.0 5.01 0.0 2.99 c
2732             0.0 0.62 1.12 -0.11 2.11 -0.11 c
2733             3.2 -0.11 4.18 0.73 4.18 1.93 c

```

```

2734         3.49 1.93 m
2735         3.49 1.07 2.83 0.5 2.11 0.5 c
2736         1.46 0.5 1.07 0.99 0.87 1.6 c
2737         0.77 1.84 0.79 2.08 0.79 2.22 c
2738         0.79 2.84 1.39 3.34 2.15 3.34 c
2739         2.96 3.34 3.49 2.67 3.49 1.93 c
2740         h f
2741     }%
2742     \hfill
2743 }%
2744 }

2745 \tlconstcn{c_@@_glyph_7_tl}
2746 {%
2747     \hbox to 5pt
2748     {%
2749         \hfill
2750         \pdfextension literal
2751         {
2752             4.36 5.8 m
2753             4.36 6.11 4.12 6.11 3.96 6.11 c
2754             0.66 6.11 l
2755             0.6 6.27 0.42 6.27 0.34 6.27 c
2756             0.0 6.27 0.0 6.05 0.0 5.86 c
2757             0.0 5.44 l
2758             0.0 5.25 0.0 5.03 0.34 5.03 c
2759             0.69 5.03 0.69 5.22 0.69 5.5 c
2760             3.33 5.5 l
2761             1.6 3.66 1.26 1.46 1.26 0.36 c
2762             1.26 0.22 1.26 -0.11 1.61 -0.11 c
2763             1.78 -0.11 1.95 0.0 1.95 0.29 c
2764             2.0 3.13 3.53 4.87 4.14 5.46 c
2765             4.34 5.64 4.36 5.66 4.36 5.8 c
2766             h f
2767         }%
2768         \hfill
2769     }%
2770 }

2771 \tlconstcn{c_@@_glyph_8_tl}
2772 {%
2773     \hbox to 5pt
2774     {%
2775         \hfill
2776         \pdfextension literal
2777         {
2778             4.36 1.74 m
2779             4.36 2.49 3.71 3.08 2.97 3.29 c
2780             3.77 3.56 4.22 4.05 4.22 4.62 c
2781             4.22 5.44 3.37 6.22 2.18 6.22 c
2782             0.98 6.22 0.14 5.43 0.14 4.62 c
2783             0.14 4.05 0.6 3.55 1.39 3.29 c
2784             0.64 3.08 0.0 2.49 0.0 1.74 c
2785             0.0 0.77 0.93 -0.11 2.18 -0.11 c
2786             3.44 -0.11 4.36 0.77 4.36 1.74 c
2787             3.53 4.61 m
2788             3.53 4.04 2.91 3.6 2.18 3.6 c
2789             1.45 3.6 0.83 4.05 0.83 4.61 c
2790             0.83 5.13 1.39 5.61 2.18 5.61 c
2791             2.96 5.61 3.53 5.13 3.53 4.61 c
2792             3.67 1.75 m
2793             3.67 1.06 3.01 0.5 2.18 0.5 c
2794             1.35 0.5 0.69 1.06 0.69 1.75 c
2795             0.69 2.36 1.27 2.99 2.18 2.99 c
2796             3.1 2.99 3.67 2.36 3.67 1.75 c

```

```

2797         h f
2798     }%
2799     \hfill
2800 }%
2801 }
2802 \tlconstcn{c_@@_glyph_9_tl}
2803 {%
2804     \hbox to 5pt
2805     {%
2806         \hfill
2807         \pdfextension literal
2808         {
2809             4.18 3.12 m
2810             4.18 5.53 3.07 6.22 2.12 6.22 c
2811             1.01 6.22 0.0 5.39 0.0 4.18 c
2812             0.0 3.04 0.88 2.15 1.98 2.15 c
2813             2.5 2.15 3.02 2.32 3.47 2.73 c
2814             3.37 1.49 2.59 0.5 1.63 0.5 c
2815             1.54 0.5 1.18 0.51 0.97 0.67 c
2816             1.01 0.72 1.06 0.78 1.06 0.95 c
2817             1.06 1.19 0.88 1.39 0.62 1.39 c
2818             0.37 1.39 0.17 1.22 0.17 0.92 c
2819             0.17 0.6 0.35 -0.11 1.63 -0.11 c
2820             2.95 -0.11 4.18 1.14 4.18 3.12 c
2821             3.4 3.89 m
2822             3.4 3.34 2.86 2.77 2.03 2.77 c
2823             1.22 2.77 0.69 3.44 0.69 4.18 c
2824             0.69 5.05 1.4 5.61 2.12 5.61 c
2825             2.55 5.61 2.83 5.38 2.99 5.18 c
2826             3.31 4.79 3.4 4.6 3.4 3.89 c
2827         h f
2828     }%
2829     \hfill
2830 }%
2831 }
2832 \ExplSyntaxOn

```

2.16 We load piton.lua

```

2833 \cs_new_protected:Npn \@@_test_version:n #1
2834 {
2835     \str_if_eq:onF \PitonFileVersion { #1 }
2836     { \@@_error:n { bad~version~of~piton.lua } }
2837 }
2838 \hook_gput_code:nnn { begindocument } { . }
2839 {
2840     \lua_load_module:n { piton }
2841     \lua_now:n
2842     {
2843         tex.sprint ( luatexbase.catcodetables.expl ,
2844                     [[\@@_test_version:n {}] .. piton_version .. "]" )
2845     }
2846 }
</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`.

```
2847  $\langle$ *LUA $\rangle$ 
2848 piton.comment_latex = piton.comment_latex or ">"
2849 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`.

```
2850 piton.write_files = { }
2851 piton.join_files = { }

2852 local sprintL3
2853 function sprintL3 ( s )
2854   tex.sprint ( luatexbase.catcodetables.expl , s )
2855 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.

```
2856 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2857 local Cg, Cmt, Cb = lpeg.Cg, lpeg.Cmt, lpeg.Cb
2858 local B, R = lpeg.B, lpeg.R
```

The following line is mandatory.

```
2859 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”).

```
2860 local Q
2861 function Q ( pattern )
2862   return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2863 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.

```
2864 local L
2865 function L ( pattern ) return
2866   Ct ( C ( pattern ) )
2867 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.

```
2868 local Lc
2869 function Lc ( string ) return
2870   Cc ( { luatexbase.catcodetables.expl , string } )
2871 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)

```

2872 local K
2873 function K ( style , pattern ) return
2874   Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
2875   * Q ( pattern )
2876   * Lc "}}"
2877 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.

```

2878 local WithStyle
2879 function WithStyle ( style , pattern ) return
2880   Ct ( Cc "Open" * Cc ( [[{\PitonStyle{ ]] .. style .. "}{" ) * Cc "}}" )
2881   * pattern
2882   * Ct ( Cc "Close" )
2883 end

```

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

```

2884 Escape = P ( false )
2885 EscapeClean = P ( false )
2886 if piton.begin_escape then
2887   Escape =
2888     P ( piton.begin_escape )
2889     * Lc [[\l_@@_after_begin_escape_tl]]
2890     * L ( ( 1 - P ( piton.end_escape ) ) ^ 0 )
2891     * Lc [[\l_@@_before_end_escape_tl]]
2892     * 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).

```

2893 EscapeClean =
2894   P ( piton.begin_escape )
2895   * ( 1 - P ( piton.end_escape ) ) ^ 0
2896   * P ( piton.end_escape )
2897 end
2898 EscapeMath = P ( false )
2899 if piton.begin_escape_math then
2900   EscapeMath =
2901     P ( piton.begin_escape_math )
2902     * Lc "$"
2903     * L ( ( 1 - P ( piton.end_escape_math ) ) ^ 1 )
2904     * Lc "$"
2905     * P ( piton.end_escape_math )
2906 end

```

The basic syntactic LPEG

```

2907 local alpha , digit = lpeg.alpha , lpeg.digit
2908 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.

```

2909 local letter = alpha + "_" + "â" + "ã" + "ç" + "ê" + "è" + "é" + "ë" + "í" + "î"
2910               + "ô" + "õ" + "ü" + "Â" + "Ã" + "Ç" + "É" + "Ê" + "Ë" + "Í"
2911               + "Î" + "Ï" + "Ô" + "Õ" + "Ü"
2912
2913 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).

```

2914 local identifier = letter * alphanum ^ 0

```

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

```

2915 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

```

2916 local allow_underscores_except_first
2917 function allow_underscores_except_first ( p )
2918     return p * ( P "_" + p ) ^ 0
2919 end
2920 local allow_underscores
2921 function allow_underscores ( p )
2922     return ( P "_" + p ) ^ 0
2923 end
2924 local digits_to_number
2925 function digits_to_number(prefix, digits)
2926     -- The edge cases of what is allowed in number literals is modelled after
2927     -- OCaml numbers, which seems to be the most permissive language
2928     -- in this regard (among C, OCaml, Python & SQL).
2929     return prefix
2930         * allow_underscores_except_first(digits^1)
2931         * ( P "." * #(1 - P ".") * allow_underscores(digits) ) ^ -1
2932         * ( S "eE" * S "+-" ^ -1 * allow_underscores_except_first(digits^1) ) ^ -1
2933 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).

```

2934 local Number =
2935     K ( 'Number.Internal' ,
2936         digits_to_number ( P "0x" + P "0X", R "af" + R "AF" + digit )
2937         + digits_to_number ( P "0o" + P "0O", R "07" )
2938         + digits_to_number ( P "0b" + P "0B", R "01" )
2939         + digits_to_number ( "" , digit )
2940     )

```

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?).

```

2941 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`.

```

2942 if piton.begin_escape then
2943   lpeg_central = lpeg_central - piton.begin_escape
2944 end
2945 if piton.begin_escape_math then
2946   lpeg_central = lpeg_central - piton.begin_escape_math
2947 end
2948 local Word = Q ( lpeg_central ^ 1 )

2949 local Space = Q " " ^ 1
2950 local SkipSpace = Q " " ^ 0
2951
2952 local Punct = Q ( S ".,:;!" )
2953
2954 local Tab = "\t" * Lc [[ \@_tab: ]]

2955 local LeadingSpace = Lc [[ \@_leading_space: ]] * P " "

2956 local Delim = K ( 'Delim' , 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.

```

2957 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).

```

2958 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
2959 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
2960 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
2961 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.

```

2962 local detectedCommands = P ( false )
2963 for _ , x in ipairs ( detected_commands ) do
2964   detectedCommands = detectedCommands + P ( "\\" .. x )
2965 end

```

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

```

2966 local rawDetectedCommands = P ( false )
2967 for _ , x in ipairs ( raw_detected_commands ) do
2968   rawDetectedCommands = rawDetectedCommands + P ( "\\" .. x )
2969 end

```

```

2970 local beamerCommands = P ( false )
2971 for _ , x in ipairs ( beamer_commands ) do
2972     beamerCommands = beamerCommands + P ( "\\\" .. x )
2973 end
2974
2974 local beamerEnvironments = P ( false )
2975 for _ , x in ipairs ( beamer_environments ) do
2976     beamerEnvironments = beamerEnvironments + P ( x )
2977 end

```

Several tools for the construction of the main LPEG

```

2978 local LPEG0 = { }
2979 local LPEG1 = { }
2980 local LPEG2 = { }
2981 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*.

```

2982 local Compute_braces
2983 function Compute_braces ( lpeg_string ) return
2984     P { "E" ,
2985         E =
2986         (
2987             "{" * V "E" * "}"
2988             +
2989             lpeg_string
2990             +
2991             ( 1 - S "{" )
2992             ) ^ 0
2993     }
2994 end

```

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

```

2995 local Compute_DetectedCommands
2996 function Compute_DetectedCommands ( lang , braces ) return
2997     Ct (
2998         Cc "Open"
2999         * C ( detectedCommands * space ^ 0 * P "{" )
3000         * Cc "}"
3001     )
3002     * ( braces
3003         / ( function ( s )
3004             if s ~= '' then return
3005                 LPEG1[lang] : match ( s )
3006             end
3007         end )
3008     )
3009     * P "}"
3010     * Ct ( Cc "Close" )
3011 end
3012
3012 local Compute_RawDetectedCommands
3013 function Compute_RawDetectedCommands ( lang , braces ) return
3014     Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
3015 end

```

```

3016 local Compute_LPEG_cleaner
3017 function Compute_LPEG_cleaner ( lang , braces ) return
3018   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
3019         * ( braces
3020           / ( function ( s )
3021               if s ~= '' then return
3022                 LPEG_cleaner[lang] : match ( s )
3023             end
3024           end )
3025         )
3026       * "}"
3027     + EscapeClean
3028     + C ( P ( 1 ) )
3029     ) ^ 0 ) / table.concat
3030 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`.

```

3031 local ParseAgain
3032 function ParseAgain ( code )
3033   if code ~= '' then return

```

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

```

3034   LPEG1[piton.language] : match ( code )
3035 end
3036 end

```

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

```

3037 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.

```

3038 local Compute_Beamer
3039 function Compute_Beamer ( lang , braces )

```

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

```

3040   local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
3041   lpeg = lpeg +
3042     Ct ( Cc "Open"
3043         * C ( beamerCommands
3044             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3045             * P "{"
3046           )
3047         * Cc "}"
3048       )
3049     * ( braces /
3050       ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3051     * "}"
3052     * Ct ( Cc "Close" )

```

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

```

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

```

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

```

3061 lpeg = lpeg +
3062   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
3063   * ( braces
3064     / ( function ( s )
3065       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3066   * L ( P "}{" )
3067   * ( braces
3068     / ( function ( s )
3069       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3070   * L ( P "}{" )
3071   * ( braces
3072     / ( function ( s )
3073       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3074   * L ( P "}" )

```

Now, the environments of Beamer.

```

3075 for _ , x in ipairs ( beamer_environments ) do
3076   lpeg = lpeg +
3077     Ct ( Cc "Open"
3078       * C (
3079         P ( [[\begin{]] .. x .. "]" )
3080         * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3081       )
3082       * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
3083       * Cc ( [[\end{]] .. x .. "]" )
3084     )
3085   * (

```

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).

```

3086     (
3087       P { "E" ,
3088         E = (
3089           P ( [[\begin{]] .. x .. "]" )
3090           * V "E"
3091           * P ( [[\end{]] .. x .. "]" )
3092         +
3093         (
3094           1
3095           - P ( [[\begin{]] .. x .. "]" )
3096           - P ( [[\end{]] .. x .. "]" )
3097         )
3098       ) ^ 0
3099     }
3100   )
3101   / ( function ( s )
3102     if s ~= '' then return
3103       LPEG1[lang] : match ( s )
3104     end
3105   end )
3106 )

```

```

3107         * P ( [[\end{}} .. x .. "}" )
3108         * Ct ( Cc "Close" )
3109     end

```

Now, you can return the value we have computed.

```

3110     return lpeg
3111 end

```

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

```

3112 local CommentMath =
3113     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`.

```

3114 local Prompt =
3115     K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
3116     * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

3117 local EOL =
3118     P "\r"
3119     *
3120     (
3121         space ^ 0 * -1
3122         +
3123         Cc "EOL"
3124     )
3125     * ( LeadingSpace ^ 0 * # ( 1 - S " \r" ) ) ^ -1

```

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

```

3126 local CommentLaTeX =
3127     P ( piton.comment_latex )
3128     * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}}]
3129     * L ( ( 1 - P "\r" ) ^ 0 )
3130     * Lc "}}"
3131     * ( 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).

```

3132 --python Python
3133 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.

```

3134 local Operator =
3135     K ( 'Operator' ,
3136         P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "://" + "*"
3137         + S "--+/*%=<>&.@|" )
3138
3139 local OperatorWord =
3140     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`.

```

3141 local For = K ( 'Keyword' , P "for" )
3142         * Space
3143         * Identifier
3144         * Space
3145         * K ( 'Keyword' , P "in" )
3146
3147 local Keyword =
3148   K ( 'Keyword' ,
3149     P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
3150     "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
3151     "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
3152     "try" + "while" + "with" + "yield" + "yield from" )
3153   + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
3154
3155 local Builtin =
3156   K ( 'Name.Builtin' ,
3157     P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
3158     "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
3159     "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
3160     "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
3161     "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
3162     "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
3163     + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
3164     "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
3165     "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
3166     "vars" + "zip" )
3167
3168 local Exception =
3169   K ( 'Exception' ,
3170     P "ArithmeticError" + "AssertionError" + "AttributeError" +
3171     "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
3172     "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
3173     "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
3174     "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
3175     "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
3176     "NotImplementedError" + "OSError" + "OverflowError" +
3177     "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
3178     "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
3179     "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
3180     + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
3181     "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
3182     "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
3183     "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
3184     "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
3185     "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
3186     "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
3187     "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
3188     "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
3189     "RecursionError" )
3190
3191 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.

```

3192 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:`

```

3193 local DefClass =
3194   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`

```

3195 local ImportAs =
3196   K ( 'Keyword' , "import" )
3197   * Space
3198   * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
3199   * (
3200     ( Space * K ( 'Keyword' , "as" ) * Space
3201       * K ( 'Name.Namespace' , identifier ) )
3202     +
3203     ( SkipSpace * Q "," * SkipSpace
3204       * K ( 'Name.Namespace' , identifier ) ) ^ 0
3205   )

```

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`

```

3206 local FromImport =
3207   K ( 'Keyword' , "from" )
3208   * Space * K ( 'Name.Namespace' , identifier )
3209   * 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'''	"""text"""

We have also to deal with the interpolations in the f-strings. Here is an example of a f-string with an interpolation and a format 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).

```

3210 local PercentInterpol =
3211   K ( 'String.Interpol' ,
3212     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`.

```

3213 * ( "(" * alphanum ^ 1 * ")" ) ^ -1
3214 * ( S "-#0 +" ) ^ 0
3215 * ( digit ^ 1 + "*" ) ^ -1
3216 * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
3217 * ( S "HLL" ) ^ -1
3218 * S "sdfFeExXorgiGauc%"
3219 )

```

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

```

3220 local SingleShortString =
3221   WithStyle ( 'String.Short.Internal' ,

```

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

```

3222   Q ( P "f'" + "F'" )
3223   * (
3224     K ( 'String.Interpol' , "{" )
3225     * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0 )
3226     * Q ( P ":" * ( 1 - S "}':" ) ^ 0 ) ^ -1
3227     * K ( 'String.Interpol' , "}" )
3228     +
3229     SpaceInString
3230     +
3231     Q ( ( P "\\'" + "\\\\" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
3232   ) ^ 0
3233   * Q ""
3234   +

```

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

```

3235   Q ( P "'" + "r'" + "R'" )
3236   * ( Q ( ( P "\\'" + "\\\\" + 1 - S " 'r%" ) ^ 1 )
3237     + SpaceInString
3238     + PercentInterpol
3239     + Q "%"
3240   ) ^ 0
3241   * Q "" )
3242 local DoubleShortString =
3243   WithStyle ( 'String.Short.Internal' ,
3244     Q ( P "f\"" + "F\"" )
3245     * (
3246       K ( 'String.Interpol' , "{" )
3247       * K ( 'Interpol.Inside' , ( 1 - S "}\" )" ) ^ 0 )
3248       * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}\" )" ) ^ 0 ) ) ^ -1
3249       * K ( 'String.Interpol' , "}" )
3250     +
3251     SpaceInString
3252     +
3253     Q ( ( P "\\\"" + "\\\\" + "{{" + "}}" + 1 - S " {}\" )" ^ 1 )
3254   ) ^ 0
3255   * Q "\"
3256   +
3257   Q ( P "\" + "r\"" + "R\"" )
3258   * ( Q ( ( P "\\\"" + "\\\\" + 1 - S " \"r%" ) ^ 1 )
3259     + SpaceInString
3260     + PercentInterpol
3261     + Q "%"
3262   ) ^ 0
3263   * 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`.

```

3264
3265     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.

```

3266     local braces =
3267         Compute_braces
3268         (
3269             ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
3270             * ( P '\\\"' + 1 - S "\"" ) ^ 0 * "\""
3271         +
3272             ( P '\'' + 'r\'' + 'R\'' + 'f\'' + 'F\'' )
3273             * ( P '\\\'' + 1 - S '\'' ) ^ 0 * '\''
3274         )
3275
3276     if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

Detected commands

```

3277     DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
3278     + Compute_RawDetectedCommands ( 'python' , braces )

```

LPEG_cleaner

```

3279     LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

The long strings

```

3280     local SingleLongString =
3281         WithStyle ( 'String.Long.Internal' ,
3282             ( Q ( S "fF" * P "'''" )
3283                 * (
3284                     K ( 'String.Interpol' , "{" )
3285                     * K ( 'Interpol.Inside' , ( 1 - S "};\r" - "'''" ) ^ 0 )
3286                     * Q ( P ":" * ( 1 - S "};\r" - "'''" ) ^ 0 ) ^ -1
3287                     * K ( 'String.Interpol' , "}" )
3288                     +
3289                     Q ( ( 1 - P "'''" - S "{'}\r" ) ^ 1 )
3290                     +
3291                     EOL
3292                 ) ^ 0
3293             +
3294             Q ( ( S "rR" ) ^ -1 * "'''" )
3295             * (
3296                 Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
3297                 +
3298                 PercentInterpol
3299                 +
3300                 P "%"
3301                 +
3302                 EOL
3303             ) ^ 0
3304         )
3305     * Q "'''" )

```

```

3306 local DoubleLongString =
3307     WithStyle ( 'String.Long.Internal' ,
3308         (
3309             Q ( S "fF" * "\"\\\"" )
3310             * (
3311                 K ( 'String.Interpol', "{ " )
3312                 * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "\"\\\"" ) ^ 0 )
3313                 * Q ( ":" * (1 - S "}:\\r" - "\"\\\"" ) ^ 0 ) ^ -1
3314                 * K ( 'String.Interpol' , "}" )
3315                 +
3316                 Q ( ( 1 - S "{}\\r" - "\"\\\"" ) ^ 1 )
3317                 +
3318                 EOL
3319             ) ^ 0
3320         +
3321         Q ( S "rR" ^ -1 * "\"\\\"" )
3322         * (
3323             Q ( ( 1 - P "\"\\\"" - S "%\\r" ) ^ 1 )
3324             +
3325             PercentInterpol
3326             +
3327             P "%"
3328             +
3329             EOL
3330         ) ^ 0
3331     )
3332     * Q "\"\\\""
3333 )
3334 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`).

```

3335 local StringDoc =
3336     K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\\\"" )
3337     * ( K ( 'String.Doc.Internal' , (1 - P "\"\\\"" - "\\r" ) ^ 0 ) * EOL
3338         * Tab ^ 0
3339     ) ^ 0
3340     * 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.

```

3341 local Comment =
3342     WithStyle
3343     ( 'Comment.Internal' ,
3344         Q "#" * ( CommentMath + Q ( ( 1 - S "$\\r" ) ^ 1 ) ) ^ 0 -- $
3345     )
3346     * ( 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).

```

3347 local expression =
3348     P { "E" ,
3349         E = ( "'" * ( P "\\'" + 1 - S "'\\r" ) ^ 0 * "'"
3350             + "\\\"" * ( P "\\\"" + 1 - S "\\\"\\r" ) ^ 0 * "\\\""
3351             + "{" * V "F" * "}"
3352             + "(" * V "F" * ")" )

```

```

3353         + "[" * V "F" * "]"
3354         + ( 1 - S "{}()[]\r," ) ^ 0 ,
3355     F = (   "{" * V "F" * "}"
3356           + "(" * V "F" * ")"
3357           + "[" * V "F" * "]"
3358           + ( 1 - S "{}()[]\r\"" ) ^ 0
3359     }

```

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`.

```

3360     local Params =
3361     P { "E" ,
3362         E = ( V "F" * ( Q "," * V "F" ) ^ 0 ) ^ -1 ,
3363         F = SkipSpace * ( Identifier + Q "**args" + Q "**kwargs" ) * SkipSpace
3364           * ( Q ":" * SkipSpace * K ( 'Name.Type' , expression ) ) ^ -1 -- modified 2026/07/15
3365           * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
3366     }

```

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`...

```

3367     local DefFunction =
3368     K ( 'Keyword' , "def" )
3369     * Space
3370     * K ( 'Name.Function.Internal' , identifier )
3371     * SkipSpace
3372     * Q "(" * Params * Q ")"
3373     * SkipSpace
3374     * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3375     * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
3376     * Q ":"
3377     * ( SkipSpace
3378       * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
3379       * Tab ^ 0
3380       * SkipSpace
3381       * StringDoc ^ 0 -- there may be additional docstrings
3382     ) ^ -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

```

3383     local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

The main LPEG for the language Python

```
3384 local EndKeyword
3385     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3386     EscapeMath + -1
```

First, the main loop :

```
3387 local Main =
3388     space ^ 0 * EOL
3389     + Space
3390     + Tab
3391     + Escape + EscapeMath
3392     + Beamer
3393     + CommentLaTeX
3394     + DetectedCommands
3395     + Prompt
3396     + LongString
3397     + Comment
3398     + ExceptionInConsole
3399     + Delim
3400     + Operator
3401     + OperatorWord * EndKeyword
3402     + ShortString
3403     + Punct
3404     + FromImport
3405     + RaiseException
3406     + DefFunction
3407     + DefClass
3408     + For
3409     + Keyword * EndKeyword
3410     + Decorator
3411     + Builtin * EndKeyword
3412     + Identifier
3413     + Number
3414     + Word
```

Here, we must not put local, of course.

```
3415 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.

```
3416 LPEG2.python =
3417     Ct (
3418         ( space ^ 0 * "\r" ) ^ -1
3419         * Lc [[ \@@_begin_line: ]]
3420         * LeadingSpace ^ 0
3421         * ( space ^ 1 * -1 + Main ) ^ 0
3422         * -1
3423         * Lc [[ \@@_end_line: ]]
3424     )
```

End of the Lua scope for the language Python.

```
3425 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).

```

3426 --ocaml Ocaml OCaml
3427 do

3428     local SkipSpace = ( Q " " + EOL ) ^ 0
3429     local Space = ( Q " " + EOL ) ^ 1

3430     local braces = Compute_braces ( '\'' * ( 1 - S "\"" ) ^ 0 * '\'' )

3431     if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
3432     DetectedCommands =
3433         Compute_DetectedCommands ( 'ocaml' , braces )
3434         + Compute_RawDetectedCommands ( 'ocaml' , braces )
3435     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.

```

3436     function Q ( pattern, strict )
3437         if strict ~= nil then
3438             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3439         else
3440             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3441                 + Beamer + DetectedCommands + EscapeMath + Escape
3442         end
3443     end

3444     local K
3445     function K ( style , pattern, strict ) return
3446         Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
3447         * Q ( pattern, strict )
3448         * Lc "}"
3449     end

3450     local WithStyle
3451     function WithStyle ( style , pattern ) return
3452         Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{ ]] .. style .. "}{" ) * Cc "}" )
3453         * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
3454         * Ct ( Cc "Close" )
3455     end

```

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

```

3456     local balanced_parens =
3457         P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }

```

The strings of OCaml

```

3458     local ocaml_string =
3459         P "\""
3460         * (
3461             P " "
3462             +
3463             P ( ( P '\\\'' + 1 - S " \r" ) ^ 1 )
3464             +
3465             EOL -- ?
3466         ) ^ 0
3467         * P "\""

```

```

3468 local String =
3469   WithStyle
3470     ( 'String.Long.Internal' ,
3471       Q "\""
3472       * (
3473         SpaceInString
3474         +
3475         Q ( ( P '\\\\' + 1 - S "\\r" ) ^ 1 )
3476         +
3477         EOL
3478       ) ^ 0
3479       * Q "\""
3480     )

```

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

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

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

```

3481 local ext = ( R "az" + "_" ) ^ 0
3482 local open = "{" * Cg ( ext , 'init' ) * "/"
3483 local close = "/" * C ( ext ) * "}"
3484 local closeeq =
3485   Cmt ( close * Cb ( 'init' ) ,
3486     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3487 local QuotedStringBis =
3488   WithStyle ( 'String.Long.Internal' ,
3489     (
3490       Space
3491       +
3492       Q ( ( 1 - S "\\r" ) ^ 1 )
3493       +
3494       EOL
3495     ) ^ 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.

```

3496 local QuotedString =
3497   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3498   ( 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).

```

3499 local comment =
3500   P {
3501     "A" ,
3502     A = Q "(*"
3503       * ( V "A"
3504         + Q ( ( 1 - S "\\r\\$" - "(" - ")" ) ^ 1 ) -- $
3505         + Q ( ocaml_string )
3506         + "$" * K ( 'Comment.Math' , ( 1 - S "\\r" ) ^ 1 ) * "$" -- $
3507         + EOL
3508       ) ^ 0
3509       * Q "*)"
3510   }
3511 local Comment = WithStyle ( 'Comment.Internal' , comment )

```

Some standard LPEG

```
3512 local Delim = K ( 'Delim' , P "[|" + "|]" + S "[]" )
3513 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.

```
3514 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.

```
3515 local Constructor =
3516   P "::"
```

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

```
3517 * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3518 +
3519 P "[]"
3520 * Lc ([{\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]])
3521 K ( 'Name.Constructor' ,
3522     Q "`" ^ -1 * cap_identifier
3523     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )
```

```
3524 local ModuleType = K ( 'Name.Type' , cap_identifier )
```

```
3525 local OperatorWord =
3526   K ( 'Operator.Word' ,
3527       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )
```

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

```
3528 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3529   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3530   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3531   "struct" + "type" + "val"
```

```
3532 local Keyword =
3533   K ( 'Keyword' ,
3534       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3535       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3536       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3537       + "virtual" + "when" + "while" + "with" )
3538   + K ( 'Keyword.Constant' , P "true" + "false" )
3539   + K ( 'Keyword.Governing' , governing_keyword )
```

```
3540 local EndKeyword
3541   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3542   + EscapeMath + -1
```

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

```
3543 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3544   - ( 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`.

```
3545 local Identifier = K ( 'Identifier.Internal' , identifier )
```

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

```

3546 local ocaml_char =
3547   P "" *
3548   (
3549     ( 1 - S "\\\" )
3550     + "\\\"
3551     * ( S "\\ntbr \"
3552         + digit * digit * digit
3553         + P "x" * ( digit + R "af" + R "AF" )
3554                   * ( digit + R "af" + R "AF" )
3555                   * ( digit + R "af" + R "AF" )
3556         + P "o" * R "03" * R "07" * R "07" )
3557   )
3558   * ""
3559 local Char =
3560   K ( 'String.Short.Internal', ocaml_char )

```

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

```

3561 local TypeParameter =
3562   K ( 'TypeParameter' ,
3563       "" * 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.

```

3564 local DotNotation =
3565   ( K ( 'Name.Module' , cap_identifier )
3566     * Q "."
3567     * ( K ( 'Name.Module' , cap_identifier )
3568         * ( Q "." * K ( 'Name.Module' , cap_identifier ) ) ^ 0
3569         * ( Q "." * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ) ^ -1
3570         + Identifier + Constructor + Q "(" + Q "[" + Q "{" )
3571     + Identifier
3572     * Q "."
3573     * K ( 'Name.Field' , identifier ) )
3574   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0
3575   * ( Q "." * # Q "(" ) ^ -1

```

The records

```

3576 local expression_for_fields_type =
3577   P { "E" ,
3578       E = ( "{ " * V "F" * "}"
3579            + "(" * V "F" * ")"
3580            + TypeParameter
3581            + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
3582       F = ( "{ " * V "F" * "}"
3583            + "(" * V "F" * ")"
3584            + ( 1 - S "{}()[]\r\"" ) + TypeParameter ) ^ 0
3585   }

```

```

3586 local expression_for_fields_value =
3587   P { "E" ,
3588       E = ( "{ " * V "F" * "}"
3589            + "(" * V "F" * ")"
3590            + "[" * V "F" * "]"
3591            + ocaml_string + ocaml_char
3592            + ( 1 - S "{}()[];" ) ) ^ 0 ,
3593       F = ( "{ " * V "F" * "}"
3594            + "(" * V "F" * ")"
3595            + "[" * V "F" * "]"
3596            + ocaml_string + ocaml_char
3597            + ( 1 - S "{}()[]\\"" ) ) ^ 0
3598   }

```

```

3599 local OneFieldDefinition =
3600   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3601   * K ( 'Name.Field' , identifier ) * SkipSpace
3602   * Q ":" * SkipSpace
3603   * K ( 'TypeExpression' , expression_for_fields_type )
3604   * SkipSpace

```

```

3605 local OneField =
3606   K ( 'Name.Field' , identifier ) * SkipSpace
3607   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3608   * ( C ( expression_for_fields_value ) / ParseAgain )
3609   * SkipSpace

```

The records.

```

3610 local RecordVal =
3611   Q "{" * SkipSpace
3612   *
3613   (
3614     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3615   ) ^ -1
3616   *
3617   (
3618     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3619   )
3620   * SkipSpace
3621   * Q ";" ^ -1
3622   * SkipSpace
3623   * Comment ^ -1
3624   * SkipSpace
3625   * Q "}"
3626 local RecordType =
3627   Q "{" * SkipSpace
3628   *
3629   (
3630     OneFieldDefinition
3631     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3632   )
3633   * SkipSpace
3634   * Q ";" ^ -1
3635   * SkipSpace
3636   * Comment ^ -1
3637   * SkipSpace
3638   * Q "}"
3639 local Record = RecordType + RecordVal

```

```

3640 local Operator =
3641   P "||" *

```

Don't use \hspace instead of \kern!

```

3642   Lc([{\PitonStyle{Operator}{\kern0.1em/\kern-0.2em/\kern0.1em}}]])
3643   +
3644   K ( 'Operator' ,
3645     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3646     "//" + "*" + ";" + "->" + "+." + "-." + "*." + "/" +
3647     + S "--+/*%=<>&@|" )

```

```

3648 local Builtin =
3649   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

```

```

3650 local Exception =
3651   K ( 'Exception' ,
3652     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3653     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3654     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

3655 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).

```

3656 local pattern_part =
3657   ( 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).

```
3658 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.

```

3659   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3660   *

```

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

```

3661   (
3662     K ( 'Identifier.Internal' , identifier )
3663   +
3664     Q "(" * SkipSpace
3665     * ( C ( pattern_part ) / ParseAgain )
3666     * SkipSpace

```

Of course, the specification of type is optional.

```

3667     * ( Q ":" * #(1- P"=")
3668         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3669     ) ^ -1
3670     * Q ")"
3671   )

```

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

```

3672 local DefFunction =
3673   K ( 'Keyword.Governing' , "let open" )
3674   * Space
3675   * K ( 'Name.Module' , cap_identifier )
3676   +
3677   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3678   * Space
3679   * K ( 'Name.Function.Internal' , identifier )
3680   * Space
3681   * (

```

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).

```

3682     Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3683   +
3684     Argument * ( SkipSpace * Argument ) ^ 0
3685   * (
3686     SkipSpace
3687     * Q ":" * #( 1 - P "=" )
3688     * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3689   ) ^ -1
3690   )

```

DefModule

```

3691 local DefModule =
3692   K ( 'Keyword.Governing' , "module" ) * Space
3693   *
3694   (
3695     K ( 'Keyword.Governing' , "type" ) * Space
3696     * K ( 'Name.Type' , cap_identifier )
3697     +
3698     K ( 'Name.Module' , cap_identifier ) * SkipSpace
3699     *
3700     (
3701       Q "(" * SkipSpace
3702       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3703       * Q ":" * # ( 1 - P "=" ) * SkipSpace
3704       * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3705       *
3706       (
3707         Q "," * SkipSpace
3708         * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3709         * Q ":" * # ( 1 - P "=" ) * SkipSpace
3710         * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3711       ) ^ 0
3712       * Q ")"
3713     ) ^ -1
3714     *
3715     (
3716       Q "=" * SkipSpace
3717       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3718       * Q "("
3719       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3720       *
3721       (
3722         Q ","
3723         *
3724         K ( 'Name.Module' , cap_identifier ) * SkipSpace
3725       ) ^ 0
3726       * Q ")"
3727     ) ^ -1
3728   )
3729   +
3730   K ( 'Keyword.Governing' , P "include" + "open" )
3731   * Space
3732   * K ( 'Name.Module' , cap_identifier )

```

DefType

```

3733 local DefType =
3734   K ( 'Keyword.Governing' , "type" )
3735   * Space
3736   * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3737   * SkipSpace
3738   * ( Q "+=" + Q "=" )
3739   * SkipSpace
3740   * (
3741     RecordType
3742     +

```

The following lines are a suggestion of Y. Salmon.

```

3743   WithStyle
3744   (
3745     'TypeExpression' ,
3746     (
3747       (
3748         EOL

```

```

3749         + comment
3750         + Q ( 1
3751             - P ";;"
3752             - P "type"
3753             - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3754         )
3755     ) ^ 0
3756     *
3757     (
3758         # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3759         + Q ";;"
3760         + -1
3761     )
3762 )
3763 )
3764 )

3765 local prompt =
3766   Q "utop[" * digit^1 * Q "> "
3767 local start_of_line = P(function(subject, position)
3768   if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3769     return position
3770   end
3771   return nil
3772 end)
3773 local Prompt = #start_of_line * K( 'Prompt', prompt )
3774 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3775               * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3776               * ( K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

The main LPEG for the language OCaml

```

3777 local Main =
3778   space ^ 0 * EOL
3779   + Space
3780   + Tab
3781   + Escape + EscapeMath
3782   + Beamer
3783   + DetectedCommands
3784   + TypeParameter
3785   + String + QuotedString + Char
3786   + Comment
3787   + Prompt + Answer

```

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

```

3788   + Q "~" * Identifier * ( Q ":" ) ^ -1
3789   + Q ":" * # (1 - P ":" ) * SkipSpace
3790       * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3791   + Exception
3792   + DefType
3793   + DefFunction
3794   + DefModule
3795   + Record
3796   + Keyword * EndKeyword
3797   + OperatorWord * EndKeyword
3798   + Builtin * EndKeyword
3799   + DotNotation * EndKeyword
3800   + Constructor
3801   + Identifier
3802   + Punct
3803   + Delim -- Delim is before Operator for a correct analysis of [| et |]
3804   + Operator

```

```

3805     + Number
3806     + Word

```

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

```

3807     LPEG1.ocaml = Main ^ 0

```

```

3808     LPEG2.ocaml =
3809     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).

```

3810         (
3811         (
3812             P ":"
3813             +
3814             (
3815                 ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3816                 * Identifier
3817                 * SkipSpace
3818                 * Q ":"
3819             )
3820         )
3821         * # ( 1 - S ":@" )
3822         * SkipSpace
3823         * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 ) * -1
3824     )
3825     +
3826     (
3827         ( space ^ 0 * "\r" ) ^ -1
3828         * Lc [[ \@@_begin_line: ]]
3829         * LeadingSpace ^ 0
3830         * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
3831           + space ^ 0 * EOL
3832           + Main
3833         ) ^ 0
3834         * -1
3835         * Lc [[ \@@_end_line: ]]
3836     )
3837 )

```

End of the Lua scope for the language OCaml.

```

3838 end

```

3.6 The language C

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

```

3839 --c C c++ C++
3840 do

```

```

3841     local Delim = K ( 'Delim' , S "{[()]}")
3842     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.

```

3843     local identifier = letter * alphanum ^ 0
3844
3845     local Operator =

```

```

3846 K ( 'Operator' ,
3847     P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3848     + S "~+/*%=<>&.@|!" )
3849
3850 local Keyword =
3851     K ( 'Keyword' ,
3852         P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3853         "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3854         "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3855         "register" + "restricted" + "return" + "static" + "static_assert" +
3856         "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3857         "union" + "using" + "virtual" + "volatile" + "while"
3858     )
3859     + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3860
3861 local Builtin =
3862     K ( 'Name.Builtin' ,
3863         P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3864
3865 local Type =
3866     K ( 'Name.Type' ,
3867         P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3868         "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3869         "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3870         "void" + "wchar_t" ) * Q "*" ^ 0
3871
3872 local DefFunction =
3873     Type
3874     * Space
3875     * Q "*" ^ -1
3876     * K ( 'Name.Function.Internal' , identifier )
3877     * SkipSpace
3878     * # 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:`

```

3879 local DefClass =
3880     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).

```

3881 local Character =
3882     K ( 'String.Short' ,
3883         P [['\']] + P "'" * ( 1 - P "'" ) ^ 0 * P "'" )

```

The strings of C

```

3884 String =
3885     WithStyle ( 'String.Long.Internal' ,
3886         Q "\""
3887         * ( SpaceInString
3888             + K ( 'String.Interpol' ,
3889                 "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3890             )
3891             + Q ( ( P "\\\"" + 1 - S " \"" ) ^ 1 )
3892             ) ^ 0
3893         * Q "\""
3894     )

```

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

```

3895 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
3896 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end

3897 DetectedCommands =
3898   Compute_DetectedCommands ( 'c' , braces )
3899   + Compute_RawDetectedCommands ( 'c' , braces )

3900 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

The directives of the preprocessor

```

3901 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.

```

3902 local Comment =
3903   WithStyle ( 'Comment.Internal' ,
3904     Q "/" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3905     * ( EOL + -1 )
3906
3907 local LongComment =
3908   WithStyle ( 'Comment.Internal' ,
3909     Q "/*"
3910     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3911     * Q "*/"
3912     ) -- $

```

The main LPEG for the language C

```

3913 local EndKeyword
3914   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3915     EscapeMath + -1

```

First, the main loop :

```

3916 local Main =
3917   space ^ 0 * EOL
3918   + Space
3919   + Tab
3920   + Escape + EscapeMath
3921   + CommentLaTeX
3922   + Beamer
3923   + DetectedCommands
3924   + Preproc
3925   + Comment + LongComment
3926   + Delim
3927   + Operator
3928   + Character
3929   + String
3930   + Punct
3931   + DefFunction
3932   + DefClass
3933   + Type * ( Q "*" ^ -1 + EndKeyword )
3934   + Keyword * EndKeyword
3935   + Builtin * EndKeyword
3936   + Identifier
3937   + Number
3938   + Word

```

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

```
3939   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.

```
3940   LPEG2.c =
3941   Ct (
3942       ( space ^ 0 * P "\r" ) ^ -1
3943       * Lc [[ \@@_begin_line: ]]
3944       * LeadingSpace ^ 0
3945       * ( space ^ 1 * -1 + Main ) ^ 0
3946       * -1
3947       * Lc [[ \@@_end_line: ]]
3948   )
```

End of the Lua scope for the language C.

```
3949 end
```

3.7 The language SQL

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

```
3950 --sql SQL
3951 do

3952   local LuaKeyword
3953   function LuaKeyword ( name ) return
3954       Lc [[ {\PitonStyle{Keyword}{ }}
3955       * Q ( Cmt (
3956           C ( letter * alphanum ^ 0 ) ,
3957           function ( _ , _ , a ) return a : upper ( ) == name end
3958       )
3959   )
3960   * Lc "}"
3961 end
```

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

```
3962   local identifier =
3963       letter * ( alphanum + "-" ) ^ 0
3964       + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"
3965   local Operator =
3966       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).

```
3967   local Set
3968   function Set ( list )
3969       local set = { }
3970       for _ , l in ipairs ( list ) do set[l] = true end
3971       return set
3972   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.

```

3973 local set_keywords = Set
3974 {
3975     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3976     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3977     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3978     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3979     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3980     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3981     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3982     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3983     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3984     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3985     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3986     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
3987     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3988     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3989     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3990     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3991     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3992     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3993     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3994     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3995 }
3996
3996 local set_builtins = Set
3997 {
3998     "AVG", "COUNT", "CHAR_LENGTH", "CONCAT", "CURDATE", "CURRENT_DATE",
3999     "DATE_FORMAT", "DAY", "LOWER", "LTRIM", "MAX", "MIN", "MONTH", "NOW",
4000     "RANK", "ROUND", "RTRIM", "SUBSTRING", "SUM", "UPPER", "YEAR"
4001 }

```

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.

```

4002 local Identifier =
4003   C ( identifier ) /
4004   (
4005     function ( s )
4006       if set_keywords [ s : upper ( ) ] then return

```

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

```

4007     { [{\PitonStyle{Keyword}}{} ] } ,
4008     { luatexbase.catcodetables.other , s } ,
4009     { "}" }
4010   else
4011     if set_builtins [ s : upper ( ) ] then return
4012       { [{\PitonStyle{Name.Builtin}}{} ] } ,
4013       { luatexbase.catcodetables.other , s } ,
4014       { "}" }
4015     else return
4016       { [{\PitonStyle{Name.Field}}{} ] } ,
4017       { luatexbase.catcodetables.other , s } ,
4018       { "}" }
4019     end
4020   end
4021 end
4022 )

```

The strings of SQL

```

4023 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.

```

4024 local braces = Compute_braces ( "'" * ( 1 - P "'" ) ^ 1 * "'" )
4025 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end

4026 DetectedCommands =
4027   Compute_DetectedCommands ( 'sql' , braces )
4028   + Compute_RawDetectedCommands ( 'sql' , braces )
4029 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.

```

4030 local Comment =
4031   WithStyle ( 'Comment.Internal' ,
4032     Q "--" -- syntax of SQL92
4033     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
4034     * ( EOL + -1 )
4035
4036 local LongComment =
4037   WithStyle ( 'Comment.Internal' ,
4038     Q "/*"
4039     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
4040     * Q "*/"
4041     ) -- $

```

The main LPEG for the language SQL

```

4042 local EndKeyword
4043   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
4044     EscapeMath + -1
4045
4046 local TableField =
4047   K ( 'Name.Table' , identifier )
4048   * Q "."
4049   * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
4050
4051 local OneField =
4052   (
4053     Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
4054     +
4055     K ( 'Name.Table' , identifier )
4056     * Q "."
4057     * K ( 'Name.Field' , identifier )
4058     +
4059     K ( 'Name.Field' , identifier )
4060   )
4061   * (
4062     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
4063     ) ^ -1
4064   * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
4065
4066 local OneTable =
4067   K ( 'Name.Table' , identifier )
4068   * (
4069     Space
4070     * LuaKeyword "AS"
4071     * Space
4072     * K ( 'Name.Table' , identifier )
4073     ) ^ -1
4074
4075 local WeCatchTableNames =

```

```

4075     LuaKeyword "FROM"
4076     * ( Space + EOL )
4077     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
4078     + (
4079         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
4080         + LuaKeyword "TABLE"
4081     )
4082     * ( Space + EOL ) * OneTable
4083 local EndKeyword
4084     = Space + Punct + Delim + EOL + Beamer
4085     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

4086 local Main =
4087     space ^ 0 * EOL
4088     + Space
4089     + Tab
4090     + Escape + EscapeMath
4091     + CommentLaTeX
4092     + Beamer
4093     + DetectedCommands
4094     + Comment + LongComment
4095     + Delim
4096     + Operator
4097     + String
4098     + Punct
4099     + WeCatchTableNames
4100     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
4101     + Number
4102     + Word

```

Here, we must not put local, of course.

```

4103 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.

```

4104 LPEG2.sql =
4105     Ct (
4106         ( space ^ 0 * "\r" ) ^ -1
4107         * Lc [[ \@@_begin_line: ]]
4108         * LeadingSpace ^ 0
4109         * ( space ^ 1 * -1 + Main ) ^ 0
4110         * -1
4111         * Lc [[ \@@_end_line: ]]
4112     )

```

End of the Lua scope for the language SQL.

```

4113 end

```

3.8 The language “Minimal”

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

```

4114 --minimal Minimal
4115 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:`

```

4116 local Punct = Q ( S ",:;!\\\" )
4117
4118 local Comment =
4119   WithStyle ( 'Comment.Internal' ,
4120     Q "\""
4121     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4122     )
4123     * ( EOL + -1 )
4124
4125 local String =
4126   WithStyle ( 'String.Short.Internal' ,
4127     Q "\""
4128     * ( SpaceInString
4129       + Q ( ( P "[\"]" + 1 - S " \"" ) ^ 1 )
4130       ) ^ 0
4131     * Q "\""
4132     )

```

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

```

4133 local braces = Compute_braces ( P "\"" * ( P "\\\"" + 1 - P "\"" ) ^ 1 * "\"" )
4134
4135 if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
4136
4137 DetectedCommands =
4138   Compute_DetectedCommands ( 'minimal' , braces )
4139   + Compute_RawDetectedCommands ( 'minimal' , braces )
4140
4141 LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
4142
4143 local identifier = letter * alphanum ^ 0
4144
4145 local Identifier = K ( 'Identifier.Internal' , identifier )
4146
4147 local Delim = Q ( S "{[()]}" )
4148
4149 local Main =
4150   space ^ 0 * EOL
4151   + Space
4152   + Tab
4153   + Escape + EscapeMath
4154   + CommentLaTeX
4155   + Beamer
4156   + DetectedCommands
4157   + Comment
4158   + Delim
4159   + String
4160   + Punct
4161   + Identifier
4162   + Number
4163   + Word

```

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

```

4164 LPEG1.minimal = Main ^ 0
4165
4166 LPEG2.minimal =
4167   Ct (
4168     ( space ^ 0 * "\r" ) ^ -1
4169     * Lc [ [ @@_begin_line: ] ]
4170     * LeadingSpace ^ 0
4171     * ( space ^ 1 * -1 + Main ) ^ 0
4172     * -1

```

```

4173         * Lc [[ \@@_end_line: ]]
4174     )

```

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

```

4175 end

```

3.9 The language “Verbatim”

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

```

4176 --verbatim Verbatim
4177 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”).

```

4178     local braces =
4179         P { "E" ,
4180             E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4181         }
4182
4183     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
4184
4185     DetectedCommands =
4186         Compute_DetectedCommands ( 'verbatim' , braces )
4187         + Compute_RawDetectedCommands ( 'verbatim' , braces )
4188
4189     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

4190     local lpeg_central = 1 - S " \\r"
4191     if piton.begin_escape then
4192         lpeg_central = lpeg_central - piton.begin_escape
4193     end
4194     if piton.begin_escape_math then
4195         lpeg_central = lpeg_central - piton.begin_escape_math
4196     end
4197     local Word = Q ( lpeg_central ^ 1 )
4198
4199     local Main =
4200         space ^ 0 * EOL
4201         + Space
4202         + Tab
4203         + Escape + EscapeMath
4204         + Beamer
4205         + DetectedCommands
4206         + Q [[\]]
4207         + Word

```

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

```

4208     LPEG1.verbatim = Main ^ 0
4209
4210     LPEG2.verbatim =
4211         Ct (
4212             ( space ^ 0 * "\r" ) ^ -1
4213             * Lc [[ \@@_begin_line: ]]
4214             * LeadingSpace ^ 0
4215             * ( space ^ 1 * -1 + Main ) ^ 0
4216             * -1
4217             * Lc [[ \@@_end_line: ]]
4218         )

```

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

```

4219 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).

```

4220 --EXPL expl
4221 do
4222     local Comment =
4223         WithStyle
4224         ( 'Comment.Internal' ,
4225         Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4226         )
4227         * ( EOL + -1 )

```

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

```

4228     local analyze_cs
4229     function analyze_cs ( s )
4230         local i = s : find ( ":" )
4231         if i then

```

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

```

4232         local name = s : sub ( 2 , i - 1 )
4233         local parts = name : explode ( "_" )
4234         local module = parts[1]
4235         if module == "" then module = parts[3] end

```

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

```

4236         return
4237         { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
4238         { luatexbase.catcodetables.other , s } ,
4239         { "}" } }
4240     else
4241         local p = s : sub ( 1 , 3 )
4242         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`.

```

4243         local scope = s : sub(2,2)
4244         local parts = s : explode ( "_" )
4245         local module = parts[2]
4246         if module == "" then module = parts[3] end
4247         local type = parts[#parts]
4248         return
4249         { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
4250         { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
4251         { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,
4252         { luatexbase.catcodetables.other , s } ,
4253         { "}}}}}" }
4254     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.

```

4255         return { luatexbase.catcodetables.other , s }
4256     end
4257 end
4258 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`).

```

4259     local braces =
4260     P { "E" ,
4261     E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4262     }

```

```

4263
4264 if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
4265
4266 DetectedCommands =
4267   Compute_DetectedCommands ( 'expl' , braces )
4268   + Compute_RawDetectedCommands ( 'expl' , braces )
4269
4270 LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
4271 local control_sequence = P "\\\" * ( R "Az" + "_" + ":" + "@" ) ^ 1
4272 local ControlSequence = C ( control_sequence ) / analyze_cs
4273
4274 local def_function
4275   = P [[\cs_]]
4276   * ( P "set" + "new" )
4277   * ( P "_protected" ) ^ -1
4278   * P ":N" * ( P "p" ) ^ -1 * "n"
4279
4280 local DefFunction =
4281   C ( def_function ) / analyze_cs
4282   * Space
4283   * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
4284   * ControlSequence -- Q ( ControlSequence ) ?
4285   * Lc "}"
4286
4287 local Word = Q ( ( 1 - S " \r" ) ^ 1 )
4288
4289 local Main =
4290   space ^ 0 * EOL
4291   + Space
4292   + Tab
4293   + Escape + EscapeMath
4294   + Beamer
4295   + Comment
4296   + DetectedCommands
4297   + DefFunction
4298   + ControlSequence
4299   + Word

```

Here, we must not put local, of course.

```

4297 LPEG1.expl = Main ^ 0
4298
4299 LPEG2.expl =
4300   Ct (
4301     ( space ^ 0 * "\r" ) ^ -1
4302     * Lc [[ \@@_begin_line: ]]
4303     * LeadingSpace ^ 0
4304     * ( space ^ 1 * -1 + Main ) ^ 0
4305     * -1
4306     * Lc [[ \@@_end_line: ]]
4307   )

```

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

```

4308 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.

```

4309 function piton.Parse ( language , code )

```

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

```

4310 piton.language = language
4311 local t = LPEG2[language] : match ( code )
4312 if not t then
4313     sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
4314     return -- to exit in force the function
4315 end
4316 local left_stack = {}
4317 local right_stack = {}
4318 for _ , one_item in ipairs ( t ) do
4319     if one_item == "EOL" then
4320         for i = #right_stack, 1, -1 do
4321             tex.sprint ( right_stack[i] )
4322         end

```

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

```

4323     sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
4324     tex.sprint ( table.concat ( left_stack ) )
4325 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}`.

```

4326     if one_item[1] == "Open" then
4327         tex.sprint ( one_item[2] )
4328         table.insert ( left_stack , one_item[2] )
4329         table.insert ( right_stack , one_item[3] )
4330     else
4331         if one_item[1] == "Close" then
4332             tex.sprint ( right_stack[#right_stack] )
4333             left_stack[#left_stack] = nil
4334             right_stack[#right_stack] = nil
4335         else
4336             tex.tprint ( one_item )
4337         end
4338     end
4339 end
4340 end
4341 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).

```

4342 local my_file_lines
4343 function my_file_lines ( filename )
4344     local f = io.open ( filename , 'rb' )
4345     local s = f : read ( '*a' )
4346     f : close ( )

```

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

```

4347     return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
4348 end

```

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

```

4349 function piton.ReadFile ( name , first_line , last_line )
4350     local s = ''
4351     local i = 0
4352     for line in my_file_lines ( name ) do

```

```

4353     i = i + 1
4354     if i >= first_line then
4355         s = s .. '\r' .. line
4356     end
4357     if i >= last_line then break end
4358 end

```

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

```

4359     if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
4360         s = s : sub ( 5 , -1 )
4361     end

4362     sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { }}] )
4363     tex.sprint ( luatexbase.catcodetables.other , s )
4364     sprintL3 ( "}" )
4365 end

4366 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
4367     local s
4368     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4369     piton.GobbleParse ( lang , n , splittable , s )
4370 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 `#`.

```

4371 function piton.ParseBis ( lang , code )
4372     return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
4373 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.

```

4374 function piton.ParseTer ( lang , code )
4375     return piton.Parse
4376     (
4377         lang ,
4378         code : gsub ( [[\@@_breakable_space: ]] , ' ' )
4379     )
4380 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.

```

4381 local AutoGobbleLPEG =
4382     ( (
4383         P " " ^ 0 * "\r"

```

```

4384      +
4385      Ct ( C " " ^ 0 ) / table.getn
4386      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
4387    ) ^ 0
4388    * ( Ct ( C " " ^ 0 ) / table.getn
4389      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4390  ) / math.min

```

The following LPEG is similar but works with the tabulations.

```

4391 local TabsAutoGobbleLPEG =
4392   (
4393     (
4394       P "\t" ^ 0 * "\r"
4395       +
4396       Ct ( C "\t" ^ 0 ) / table.getn
4397       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
4398     ) ^ 0
4399     * ( Ct ( C "\t" ^ 0 ) / table.getn
4400       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4401   ) / 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).

```

4402 local EnvGobbleLPEG =
4403   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
4404   * 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.

```

4405 function piton.Gobble ( n , code )
4406   if n == 0 then return
4407     code
4408   else
4409     if n == -1 then
4410       n = AutoGobbleLPEG : match ( code )

```

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

```

4411     if tonumber(n) then else n = 0 end
4412   else
4413     if n == -2 then
4414       n = EnvGobbleLPEG : match ( code )
4415     else
4416       if n == -3 then
4417         n = TabsAutoGobbleLPEG : match ( code )
4418         if tonumber(n) then else n = 0 end
4419       end
4420     end
4421   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...

```

4422   if n == 0 then return
4423     code
4424   else return

```

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

```

4425   ( Ct (
4426     ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4427     * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4428   ) ^ 0 )
4429   / table.concat

```

```

4430         ) : match ( code )
4431     end
4432 end
4433 end

```

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

```

4434 function piton.GobbleParse ( lang , n , splittable , code )
4435     piton.ComputeLinesStatus ( code , splittable )
4436     piton.last_code = piton.Gobble ( n , code )
4437     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`.

```

4438     piton.CountLines ( piton.last_code )
4439     piton.Parse ( lang , piton.last_code )
4440     piton.join_and_write ( )
4441 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`.

```

4442 function piton.join_and_write ( )
4443     if piton.join ~= '' then
4444         if not piton.join_files [ piton.join ] then
4445             piton.join_files [ piton.join ] = piton.get_last_code ( )
4446         else
4447             if piton.join_separation == '' then
4448                 piton.join_files [ piton.join ] =
4449                     piton.join_files [ piton.join ]
4450                     .. "\r\n"
4451                 .. piton.get_last_code ( )
4452             else
4453                 piton.join_files [ piton.join ] =
4454                     piton.join_files [ piton.join ]
4455                     .. "\r\n"
4456                     .. ( piton.join_separation : gsub ( '##' , '#' ) )
4457                     .. "\r\n"
4458                     .. piton.get_last_code ( )
4459             end
4460         end
4461     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.

```

4462     if piton.write ~= '' then

```

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

```

4463         local file_name = ''
4464         if piton.path_write == '' then
4465             file_name = piton.write
4466         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.

```

4467             local attr = lfs.attributes ( piton.path_write )
4468             if attr and attr.mode == "directory" then
4469                 file_name = piton.path_write .. "/" .. piton.write
4470             else

```

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

```

4471      sprintL3 [[ \@_error_or_warning:n { InexistentDirectory } ]]
4472      end
4473      end
4474      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`).

```

4475      if not piton.write_files [ file_name ] then
4476          piton.write_files [ file_name ] = piton.get_last_code ( )
4477      else
4478          piton.write_files [ file_name ] =
4479          piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4480      end
4481      end
4482      end
4483      end

```

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

```

4484 function piton.GobbleParseNoPrint ( lang , n , code )
4485     piton.last_code = piton.Gobble ( n , code )
4486     piton.last_language = lang
4487     piton.join_and_write ( )
4488 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).

```

4489 function piton.GobbleSplitParse ( lang , n , splittable , code )
4490     local chunks
4491     chunks =
4492     (
4493         Ct (
4494             (
4495                 P " " ^ 0 * "\r"
4496                 +
4497                 C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4498                     - ( P " " ^ 0 * ( P "\r" + -1 ) )
4499                 ) ^ 1
4500             )
4501         ) ^ 0
4502     )
4503     ) : match ( piton.Gobble ( n , code ) )
4504     sprintL3 [[ \begingroup ]]
4505     sprintL3
4506     (
4507         [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
4508         .. "language = " .. lang .. ","
4509         .. "splittable = " .. splittable .. "}"
4510     )
4511     for k , v in pairs ( chunks ) do
4512         if k > 1 then
4513             sprintL3 ( [[ \l_@_split_separation_tl ]] )
4514         end
4515         tex.print
4516         (
4517             [[\begin{}} .. piton.env_used_by_split .. "}" .. "\r"

```

```

4518         .. v
4519         .. [[\end{}} .. piton.env_used_by_split .. "}\r"
4520     )
4521 end
4522 sprintL3 [[ \endgroup ]]
4523 end

4524 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4525     local s
4526     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4527     piton.GobbleSplitParse ( lang , n , splittable , s )
4528 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.

```

4529 piton.string_between_chunks =
4530 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4531 .. [[ \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.

```

4532 function piton.get_last_code ( )
4533     return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4534     : gsub ( '\r\n?' , '\n' )
4535 end

```

3.14 To count the number of lines

```

4536 local CountBeamerEnvironments
4537 function CountBeamerEnvironments ( code ) return
4538 (
4539     Ct (
4540         (
4541             P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4542             +
4543             ( 1 - P "\r" ) ^ 0 * "\r"
4544         ) ^ 0
4545         * ( 1 - P "\r" ) ^ 0
4546         * -1
4547     ) / table.getn
4548 ) : match ( code )
4549 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.

```

4550 function piton.CountLines ( code )
4551     local count
4552     count =
4553         ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4554             *
4555             (
4556                 space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4557                 + space ^ 0
4558             ) ^ -1
4559             * -1

```

```

4560         ) / table.getn
4561     ) : match ( code )
4562     if piton.beamer then
4563         count = count - 2 * CountBeamerEnvironments ( code )
4564     end
4565     sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4566 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).

```

4567 function piton.CountNonEmptyLines ( code )
4568     local count = 0

```

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

```

4569     count =
4570         ( Ct ( ( P " " ^ 0 * "\r"
4571             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4572             * ( 1 - P "\r" ) ^ 0
4573             * -1
4574             ) / table.getn
4575         ) : match ( code )
4576     count = count + 1
4577     if piton.beamer then
4578         count = count - 2 * CountBeamerEnvironments ( code )
4579     end
4580     sprintL3
4581     ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4582 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.

```

4583 function piton.ComputeRange ( s , t , file_name )
4584     local first_line = -1
4585     local count = 0
4586     local last_found = false
4587     for line in io.lines ( file_name ) do
4588         if first_line == -1 then
4589             if line : sub ( 1 , #s ) == s then
4590                 first_line = count
4591             end
4592         else
4593             if line : sub ( 1 , #t ) == t then
4594                 last_found = true
4595                 break
4596             end
4597         end
4598         count = count + 1
4599     end
4600     if first_line == -1 then
4601         sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4602     else
4603         if not last_found then
4604             sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4605         end
4606     end
4607     sprintL3 (
4608         [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 '
4609         .. [[ \global \l_@@_last_line_int = ]] .. count )
4610 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`.

```
4611 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).

```
4612     local lpeg_line_beamer
4613     if piton.beamer then
4614         lpeg_line_beamer =
4615             space ^ 0
4616             * P [[\begin{}} * beamerEnvironments * "]"
4617             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4618             +
4619             space ^ 0
4620             * P [[\end{}} * beamerEnvironments * "]"
4621     else
4622         lpeg_line_beamer = P ( false )
4623     end
4624     local lpeg_empty_lines =
4625         Ct (
4626             ( lpeg_line_beamer * "\r"
4627             +
4628             P " " ^ 0 * "\r" * Cc ( 0 )
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
4636     local lpeg_all_lines =
4637         Ct (
4638             ( lpeg_line_beamer * "\r"
4639             +
4640             ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4641             ) ^ 0
4642             *
4643             ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4644         )
4645         * -1
```

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

```
4646     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`).

```

4647 local lines_status
4648 local s = splittable
4649 if splittable < 0 then s = - splittable end
4650
4651 if splittable > 0 then
4652   lines_status = lpeg_all_lines : match ( code )
4653 else

```

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

```

4653   lines_status = lpeg_empty_lines : match ( code )
4654   for i , x in ipairs ( lines_status ) do
4655     if x == 0 then
4656       for j = 1 , s - 1 do
4657         if i + j > #lines_status then break end
4658         if lines_status[i+j] == 0 then break end
4659         lines_status[i+j] = 2
4660       end
4661       for j = 1 , s - 1 do
4662         if i - j == 1 then break end
4663         if lines_status[i-j-1] == 0 then break end
4664         lines_status[i-j-1] = 2
4665       end
4666     end
4667   end
4668 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.

```

4669   for j = 1 , s - 1 do
4670     if j > #lines_status then break end
4671     if lines_status[j] == 0 then break end
4672     lines_status[j] = 2
4673   end

```

Now, from the end of the code.

```

4674   for j = 1 , s - 1 do
4675     if #lines_status - j == 0 then break end
4676     if lines_status[#lines_status - j] == 0 then break end
4677     lines_status[#lines_status - j] = 2
4678   end

```

```

4679   piton.lines_status = lines_status
4680 end

```

```

4681 function piton.TranslateBeamerEnv ( code )
4682   local s
4683   s =
4684   (
4685     Ct (
4686       (
4687         space ^ 0
4688         * C (
4689           ( P "\\begin{" + "\\end{" )
4690           * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * "\r"
4691         )
4692         + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4693       ) ^ 0
4694     *
4695     (

```

```

4696      (
4697          space ^ 0
4698          * C (
4699              ( P "\\begin{" + "\\end{" )
4700              * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * -1
4701          )
4702          + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4703      ) ^ -1
4704  )
4705  ) ^ -1 / table.concat
4706  ) : match ( code )
4707  sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4708  tex.sprint ( luatexbase.catcodetables.other , s )
4709  sprintL3 ( "]" )
4710 end

```

3.16 To create new languages with the syntax of listings

```

4711 function piton.new_language ( lang , definition )
4712     lang = lang : lower ( )

4713     local alpha , digit = lpeg.alpha , lpeg.digit
4714     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`.

```

4715     function add_to_letter ( c )
4716         if c ~= " " then table.insert ( extra_letters , c ) end
4717     end

```

For the digits, it's straitforward.

```

4718     function add_to_digit ( c )
4719         if c ~= " " then digit = digit + c end
4720     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 `}`).

```

4721     local other = S " :_@+~*/<>!?.() [] ~^=#&\"'\\$" --
4722     local extra_others = { }
4723     function add_to_other ( c )
4724         if c ~= " " then

```

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

```

4725         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 `</...>`.

```

4726         other = other + P ( c )
4727     end
4728 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`.

```

4729     local def_table
4730     if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4731         def_table = {}
4732     else
4733         local strict_braces =

```

```

4734     P { "E" ,
4735         E = ( "{" * V "F" * "}" + ( 1 - S ",{" }" ) ) ^ 0 ,
4736         F = ( "{" * V "F" * "}" + ( 1 - S "{" }" ) ) ^ 0
4737     }
4738     local cut_definition =
4739     P { "E" ,
4740         E = Ct ( V "F" * ( "," * V "F" ) ^ 0 ) ,
4741         F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4742             * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4743     }
4744     def_table = cut_definition : match ( definition )
4745 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.).

```

4746     local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4747     local tex_arg = tex_braced_arg + C ( 1 )
4748     local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4749     local args_for_tag
4750     = tex_option_arg
4751     * space ^ 0
4752     * tex_arg
4753     * space ^ 0
4754     * tex_arg
4755     local args_for_morekeywords
4756     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4757     * space ^ 0
4758     * tex_option_arg
4759     * space ^ 0
4760     * tex_arg
4761     * space ^ 0
4762     * ( tex_braced_arg + Cc ( nil ) )
4763     local args_for_moredelims
4764     = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4765     * args_for_morekeywords
4766     local args_for_morecomment
4767     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4768     * space ^ 0
4769     * tex_option_arg
4770     * space ^ 0
4771     * 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`.

```

4772     local sensitive = true
4773     local style_tag , left_tag , right_tag
4774     for _ , x in ipairs ( def_table ) do
4775         if x[1] == "sensitive" then
4776             if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4777                 sensitive = true
4778             else
4779                 if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4780             end
4781         end
4782         if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4783         if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4784         if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4785         if x[1] == "tag" then

```

```

4786     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4787     style_tag = style_tag or [[\PitonStyle{Tag}]]
4788 end
4789 end

```

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

```

4790 local Number =
4791   K ( 'Number.Internal' ,
4792     ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4793       + digit ^ 0 * "." * digit ^ 1
4794       + digit ^ 1 )
4795     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4796     + digit ^ 1
4797   )
4798 local string_extra_letters = ""
4799 for _ , x in ipairs ( extra_letters ) do
4800   if not ( extra_others[x] ) then
4801     string_extra_letters = string_extra_letters .. x
4802   end
4803 end
4804 local letter = alpha + S ( string_extra_letters )
4805   + P "â" + "à" + "ç" + "ê" + "ë" + "ê" + "ï" + "î"
4806   + "ô" + "û" + "ü" + "â" + "ã" + "ç" + "é" + "è" + "ê" + "ë"
4807   + "ï" + "î" + "ô" + "û" + "ü"
4808 local alphanum = letter + digit
4809 local identifier = letter * alphanum ^ 0
4810 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.

```

4811 local split_clist =
4812   P { "E" ,
4813     E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4814       * ( P "{" ) ^ 1
4815       * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4816       * ( P "}" ) ^ 1 * space ^ 0 ,
4817     F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4818   }

```

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

```

4819 local keyword_to_lpeg
4820 function keyword_to_lpeg ( name ) return
4821   Q ( Cmt (
4822     C ( identifier ) ,
4823     function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4824   )
4825   )
4826 end
4827 local Keyword = P ( false )
4828 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`.

```

4830 local keywords_prefix = ''

```

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

```

4831 for _ , x in ipairs ( def_table )
4832 do if x[1] == "morekeywords"
4833   or x[1] == "otherkeywords"
4834   or x[1] == "moredirectives"
4835   or x[1] == "moretexcs"
4836 then
4837   local keywords = P ( false )

```

```

4838     local style = [[\PitonStyle{Keyword}]]
4839     if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
4840     style = tex_option_arg : match ( x[2] ) or style
4841     local n = tonumber ( style )
4842     if n then
4843         if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4844     end
4845     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4846         if x[1] == "moretexcs" then
4847             keywords = Q ( [[\]] .. word ) + keywords
4848         else
4849             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*.

```

4850         then keywords = Q ( word ) + keywords
4851         else keywords = keyword_to_lpeg ( word ) + keywords
4852     end
4853 end
4854 end
4855 Keyword = Keyword +
4856     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4857 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.

```

4858     if x[1] == "keywordsprefix" then
4859         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4860         PrefixedKeyword = PrefixedKeyword
4861             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4862         keywords_prefix = keywords_prefix .. prefix
4863     end
4864 end

```

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

```

4865     local long_string = P ( false )
4866     local Long_string = P ( false )
4867     local LongString = P ( false )
4868     local central_pattern = P ( false )
4869     for _ , x in ipairs ( def_table ) do
4870         if x[1] == "morestring" then
4871             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4872             arg2 = arg2 or [[\PitonStyle{String.Long}]]
4873             if arg1 ~= "s" then
4874                 arg4 = arg3
4875             end
4876             central_pattern = 1 - S ( " \r" .. arg4 )
4877             if arg1 : match "b" then
4878                 central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4879             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...

```

4880         if arg1 : match "d" or arg1 == "m" then

```

```

4881     central_pattern = P ( arg3 .. arg3 ) + central_pattern
4882 end
4883 if arg1 == "m"
4884 then prefix = B ( 1 - letter - ")" - "]" )
4885 else prefix = P ( true )
4886 end

```

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

```

4887     long_string = long_string +
4888         prefix
4889         * arg3
4890         * ( space + central_pattern ) ^ 0
4891         * arg4

```

Now a pattern *with captures*.

```

4892     local pattern =
4893         prefix
4894         * Q ( arg3 )
4895         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4896         * Q ( arg4 )

```

We will need Long_string in the nested comments.

```

4897     Long_string = Long_string + pattern
4898     LongString = LongString +
4899         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4900         * pattern
4901         * Ct ( Cc "Close" )
4902 end
4903 end

```

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

```

4904     local braces = Compute_braces ( long_string )
4905     if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4906
4907     DetectedCommands =
4908         Compute_DetectedCommands ( lang , braces )
4909         + Compute_RawDetectedCommands ( lang , braces )
4910
4911     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*).

```

4912     local sComment = P ( false )

```

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

```

4913     local lComment = P ( false )

```

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

```

4914     local fComment = P ( false )
4915     local fComment = P ( false )
4916
4917     for _ , x in ipairs ( def_table ) do
4918         if x[1] == "morecomment" then
4919             local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4920             arg2 = arg2 or [[\PitonStyle{Comment}]]

```

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

```

4921         if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4922         if arg1 : match "l" or arg1 : match "f" then
4923             local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4924             : match ( other_args )

```

```

4925 if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4926 if arg3 == [[\%]] then arg3 = "%" end -- mandatory
4927 local MyLPEG =
4928   Ct ( Cc "Open"
4929     * Cc ( "{" .. arg2 .. [[{\PitonSpaceSubstitute{}}] ]
4930     * Cc "}" }"
4931   )
4932   * Q ( arg3 )
4933   * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $ noqa
4934   * Ct ( Cc "Close" )
4935   * ( EOL + -1 )
4936 if arg1 : match "l" then
4937   lComment = lComment + MyLPEG
4938 else
4939   fComment = fComment + MyLPEG
4940 end
4941 else
4942   local arg3 , arg4 =
4943     ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
4944   if arg1 : match "s" then
4945     sComment = sComment +
4946       Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4947       * Q ( arg3 )
4948       * (
4949         CommentMath
4950         + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
4951         + EOL
4952       ) ^ 0
4953       * Q ( arg4 )
4954       * Ct ( Cc "Close" )
4955   end
4956   if arg1 : match "n" then
4957     sComment = sComment +
4958       Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4959       * P { "A" ,
4960         A = Q ( arg3 )
4961         * ( V "A"
4962           + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
4963             - S "\r$" ) ^ 1 ) -- $
4964           + long_string
4965           + "$" -- $
4966           * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
4967           * "$" -- $
4968           + EOL
4969         ) ^ 0
4970         * Q ( arg4 )
4971       }
4972       * Ct ( Cc "Close" )
4973   end
4974 end
4975 end

```

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

```

4976 if x[1] == "moredelim" then
4977   local arg1 , arg2 , arg3 , arg4 , arg5
4978   = args_for_moredelims : match ( x[2] )
4979   local MyFun = Q
4980   if arg1 == "*" or arg1 == "**" then
4981     function MyFun ( x )
4982       if x ~= '' then return
4983         LPEG1[lang] : match ( x )
4984       end
4985     end
4986   end

```

```

4987     local left_delim
4988     if arg2 : match "i" then
4989         left_delim = P ( arg4 )
4990     else
4991         left_delim = Q ( arg4 )
4992     end
4993     if arg2 : match "l" then
4994         sComment = sComment +
4995             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4996             * left_delim
4997             * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4998             * Ct ( Cc "Close" )
4999             * ( EOL + -1 )
5000     end
5001     if arg2 : match "s" then
5002         local right_delim
5003         if arg2 : match "i" then
5004             right_delim = P ( arg5 )
5005         else
5006             right_delim = Q ( arg5 )
5007         end
5008         sComment = sComment +
5009             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
5010             * left_delim
5011             * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
5012             * right_delim
5013             * Ct ( Cc "Close" )
5014     end
5015 end
5016 end
5017
5018 local Delim = Q ( S "{[()]}")
5019 local Punct = Q ( S "=",:;!\\'\"" )
5020
5021 local EOL =
5022     P "\r"
5023     *
5024     (
5025         space ^ 0 * -1
5026         +
5027         Cc "EOL"
5028     )
5029     * ( ( 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).

```

5029     local lpeg_central = 1 - S " '\r[{}]" - digit - S ( keywords_prefix )
5030     if piton.begin_escape then
5031         lpeg_central = lpeg_central - piton.begin_escape
5032     end
5033     if piton.begin_escape_math then
5034         lpeg_central = lpeg_central - piton.begin_escape_math
5035     end
5036     local Word = Q ( lpeg_central ^ 1 )
5037
5038     local Main =
5039         space ^ 0 * EOL * ( fComment + lComment ) ^ 0
5040         + Space
5041         + Tab
5042         + Escape + EscapeMath
5043         + CommentLaTeX
5044         + Beamer
5045         + DetectedCommands

```

```

5045     + sComment
5046     + lComment

```

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

```

5047     + LongString
5048     + Delim
5049     + PrefixedKeyword
5050     + Keyword * ( -1 + # ( 1 - alphanum ) )
5051     + Punct
5052     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5053     + Number
5054     + 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.

```

5055   LPEG1[lang] = Main ^ 0

```

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

```

5056   LPEG2[lang] =
5057     Ct (
5058       ( space ^ 0 * P "\r" ) ^ -1
5059       * Lc [[ \@@_begin_line: ]]
5060       * ( fComment + lComment ) ^ 0 -- 2026-05-12
5061       * LeadingSpace ^ 0
5062       * ( space ^ 1 * -1 + Main ) ^ 0
5063       * -1
5064       * Lc [[ \@@_end_line: ]]
5065     )

```

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

```

5066   if left_tag then
5067     local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
5068       * Q ( left_tag * other ^ 0 ) -- $
5069       * ( ( 1 - P ( right_tag ) ) ^ 0 )
5070       / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
5071     * Q ( right_tag )
5072     * Ct ( Cc "Close" )
5073   MainWithoutTag
5074     = space ^ 1 * -1
5075     + space ^ 0 * EOL
5076     + Space
5077     + Tab
5078     + Escape + EscapeMath
5079     + CommentLaTeX
5080     + Beamer
5081     + DetectedCommands
5082     + sComment
5083     + Delim
5084     + LongString
5085     + PrefixedKeyword
5086     + Keyword * ( -1 + # ( 1 - alphanum ) )
5087     + Punct
5088     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5089     + Number
5090     + Word
5091   LPEG0[lang] = MainWithoutTag ^ 0
5092   local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
5093     + Beamer + DetectedCommands + sComment + Tag
5094   MainWithTag
5095     = space ^ 1 * -1
5096     + space ^ 0 * EOL
5097     + Space

```

```

5098         + LPEGaux
5099         + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
5100 LPEG1[lang] = MainWithTag ^ 0
5101 LPEG2[lang] =
5102     Ct (
5103         ( space ^ 0 * P "\r" ) ^ -1
5104         * Lc [[ \@@_begin_line: ]]
5105         * Beamer
5106         * LeadingSpace ^ 0
5107         * LPEG1[lang]
5108         * -1
5109         * Lc [[ \@@_end_line: ]]
5110     )
5111 end
5112 end

```

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

```

5113 function piton.write_files_now ( )
5114     for file_name , file_content in pairs ( piton.write_files ) do
5115         local file = io.open ( file_name , "w" )
5116         if file then
5117             file : write ( file_content )
5118             file : close ( )
5119         else
5120             sprintL3
5121             ( [[ \@@_error_or_warning:nn { FileError } { ]] .. file_name .. "]" )
5122         end
5123     end
5124 end

```

3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

5125 function piton.utf16 ( str )
5126     local hex = { "FEFF" } -- BOM UTF-16BE
5127     for _, codepoint in utf8.codes(str) do
5128         table.insert(hex, string.format("%04X", codepoint))
5129     end
5130     return table.concat(hex)
5131 end
5132 </LUA>

```