

# The package `piton`\*

F. Pantigny  
fpantigny@wanadoo.fr

May 25, 2026

## Abstract

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

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

## 1 Introduction

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

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

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

Consider, for example, the following Python code:

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

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

---

\*This document corresponds to the version 4.13 of `piton`, at the date of 2026/05/25.

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

```

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

```

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

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

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

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

```

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

```

## 2 The L3 part of the implementation

### 2.1 Declaration of the package

```

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

```

```

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

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

```

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

```

20 \RequirePackage { amstext }

```

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

```

21 \RequirePackage { marginalia }

```

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

```

22 \RequirePackage { transparent }

```

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

```

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

67 \RequirePackage { luacode }

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

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

```

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

```

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

```

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

```

100 \ProcessKeyOptions

```

```

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

114 \RequirePackage { xcolor }

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

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

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

```

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

```

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

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

```

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

```

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

```

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

## 2.2 Parameters and technical definitions

```
155 \dim_new:N \l_@@_rounded_corners_dim
156 \bool_new:N \l_@@_in_label_bool
157 \dim_new:N \l_@@_tmpc_dim
```

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

```
158 \tl_new:N \l_@@_listing_tl
```

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

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

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

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

```
160 \str_new:N \l_piton_language_str
161 \str_set:Nn \l_piton_language_str { python }
```

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

```
162 \tl_new:N \g_piton_last_code_tl
```

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

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

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

```
164 \seq_new:N \g_@@_join_seq
165 \str_new:N \l_@@_join_str
```

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

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

The following parameter corresponds to the key `box`.

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

In order to have a better control over the keys.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
176 \str_new:N \l_@@_begin_range_str
177 \str_new:N \l_@@_end_range_str
```

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

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

The argument of `\PitonInputFile`.

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

The following line can't be deleted.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
187 \dim_new:N \l_@@_width_dim
```

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

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

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

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

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

It will be computed in `@@_compute_code_width:`

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

---

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

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

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

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

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

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

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

```
195 \bool_new:N \l_@@_lmodern_drawn_bool
```

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

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

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

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

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

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

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

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

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

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

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



```

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

```

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

```

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

```

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

```

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

```

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

```

263 \NewDocumentCommand { \@@_noop_rowcolor } { o m } { }

```

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

These macros must *not* be protected.

```

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

```

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

```

266 \tl_new:N \g_@@_after_line_tl

```

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

```

267 \cs_new_protected:Npn \@@_trailing_space: { }

```

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

```

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

270 \bool_new:N \g_@@_rowcolor_inside_bool

```

## 2.3 Detected commands

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

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

For each of those keys, we keep a clist of the names of such detected commands and environments. For the commands, the corresponding clist will contain the name of the commands *wihtout* the backlash.

```

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

```

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

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

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

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

```

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

```

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

```

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

```

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

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

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

```

290 \tl_new:N \g_@@_def_vertical_commands_tl

```

The following commands must *not* be protected.

```

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

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

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

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

```

## 2.4 Treatment of a line of code

```

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

```

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

```

357   \bool_if:NT \l_@@_break_lines_in_Piton_bool
358   {
359     \tl_if_eq:NnF \l_@@_space_in_string_tl { }
360     { \tl_set_eq:NN \l_@@_space_in_string_tl \@@_breakable_space: }

```

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

The first implementation was using `\tl_regex_replace_all:nnN`

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

but that programming was certainly slow.

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

```

361   \tl_replace_all:Nvn \l_tmpa_tl
362   \c_catcode_other_space_tl
363   \@@_breakable_space:
364 }
365 }
366 \l_tmpa_tl
367 }
368 \cs_generate_variant:Nn \@@_replace_spaces:n { o }

```

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

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

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

```

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

```

```

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

```

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

```

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

```

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

```

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

```

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

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

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

```

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

```

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

```

475         \group_begin:
476         \@@_compute_and_set_color:

```

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

```

477         \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
478         \bool_if:NT \g_@@_next_color_is_none_bool
479         { \dim_sub:Nn \l_tmpa_dim { 2.5 pt } }

```

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

```

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

```

Now, the colored panel.

```

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

```

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

```

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

```

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

```

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

```

The group is for the color of the background.

```

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

```

```

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

```

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

The command \@@\_compute\_and\_set\_color: sets the current color but also sets the booleans \g\_@@\_color\_is\_none\_bool and \g\_@@\_next\_color\_is\_none\_bool. It uses the current value of \l\_@@\_bg\_color\_clist, the value of \g\_@@\_line\_int (the number of the current line) but also potential token lists of the form \g\_@@\_color\_12\_tl if the end user has used the command \rowcolor.

```

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

```

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

```

551   \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
552   {
553     \tl_set_eq:Nc \l_tmpa_tl { g_@@_color_ \int_use:N \g_@@_line_int _ tl}

```

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

```

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

```

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

```

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

```

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

```

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

```



```

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

```

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

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

```

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

```

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

```

595   \int_gincr:N \g_@@_line_int

```

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

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

```

596   \par

```

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

```

597   \kern -2.5 pt

```

Now, we control page breaks after the paragraph.

```

598   \@@_add_penalty_for_the_line:
599 }

```

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

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

```

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

```

```

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

```

## 2.5 PitonOptions

```

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

639 \keys_define:nn { PitonOptions / marker }
640 {
641   beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
642   beginning .value_required:n = true ,
643   end .cs_set:Np = \@@_marker_end:n #1 ,
644   end .value_required:n = true ,
645   include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
646   unknown .code:n = \@@_error:n { Unknown-key-for-marker }
647 }

648 \keys_define:nn { PitonOptions / line-numbers }
649 {
650   true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
651   false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
652
653   start .code:n =
654     \bool_set_true:N \l_@@_line_numbers_bool
655     \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
656   start .value_required:n = true ,
657
658   skip-empty-lines .code:n =
659     \bool_if:NF \l_@@_in_PitonOptions_bool
660     { \bool_set_true:N \l_@@_line_numbers_bool }
661     \str_if_eq:nnTF { #1 } { false }
662     { \bool_set_false:N \l_@@_skip_empty_lines_bool }
663     { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
664
665   label-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_@@_label_empty_lines_bool }
670     { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
671
672   absolute .code:n =
673     \bool_if:NTF \l_@@_in_PitonOptions_bool
674     { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
675     { \bool_set_true:N \l_@@_line_numbers_bool }
676     \bool_if:NT \l_@@_in_PitonInputFile_bool

```

```

677     {
678         \bool_set_true:N \l_@@_line_numbers_absolute_bool
679         \bool_set_false:N \l_@@_skip_empty_lines_bool
680     } ,
681     absolute .value_forbidden:n = true ,
682
683     resume .code:n =
684         \bool_set_true:N \l_@@_resume_bool
685         \bool_if:NF \l_@@_in_PitonOptions_bool
686         { \bool_set_true:N \l_@@_line_numbers_bool } ,
687     resume .value_forbidden:n = true ,
688
689     sep .dim_set:N = \l_@@_numbers_sep_dim ,
690     sep .value_required:n = true ,
691     sep .initial:n = 0.7 em ,
692
693     step .int_set:N = \l_@@_numbers_step_int ,
694     step .initial:n = 1 ,
695     step .value_required:n = true ,
696
697     position .choices:nn = { left , right }
698     { \str_set_eq:NN \l_@@_line_numbers_position_str \l_keys_choice_tl } ,
699     position .value_required:n = true ,
700
701     format .tl_set:N = \l_@@_line_numbers_format_tl ,
702     format .value_required:n = true ,
703
704     lmodern10-drawn .bool_set:N = \l_@@_lmodern_drawn_bool ,
705     lmodern10-drawn .default:n = true ,
706
707     unknown .code:n =
708         \@@_unknown_key:nn
709         { PitonOptions / line-numbers }
710         { Unknown~key~for~line-numbers }
711
712 }

```

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

```

713 \keys_define:nn { PitonOptions }
714 {
715     indentations-for-Foxit .choices:nn = { true , false }
716     {
717         \tl_if_eq:VnTF \l_keys_value_tl { true }
718         { \@@_define_leading_space_Foxit: }
719         { \@@_define_leading_space_normal: }
720     } ,
721     box .choices:nn = { c , t , b , m }
722     { \str_set_eq:NN \l_@@_box_str \l_keys_choice_tl } ,
723     box .default:n = c ,
724     break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
725     break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,

```

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

```

726     detected-commands .code:n = \@@_detected_commands:n { #1 } ,
727     detected-commands .value_required:n = true ,
728     detected-commands .usage:n = preamble ,
729     vertical-detected-commands .code:n = \@@_vertical_detected_commands:n { #1 } ,
730     vertical-detected-commands .value_required:n = true ,
731     vertical-detected-commands .usage:n = preamble ,
732     raw-detected-commands .code:n = \@@_raw_detected_commands:n { #1 } ,
733     raw-detected-commands .value_required:n = true ,
734     raw-detected-commands .usage:n = preamble ,
735     detected-beamer-commands .code:n =

```

```

736 \@@_error_if_not_in_beamer:
737 \clist_put_right:Nn \l_@@_beamer_commands_clist { #1 } ,
738 detected-beamer-commands .value_required:n = true ,
739 detected-beamer-commands .usage:n = preamble ,
740 detected-beamer-environments .code:n =
741 \@@_error_if_not_in_beamer:
742 \clist_put_right:Nn \l_@@_beamer_environments_clist { #1 } ,
743 detected-beamer-environments .value_required:n = true ,
744 detected-beamer-environments .usage:n = preamble ,

```

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

```

745 begin-escape .code:n =
746 \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
747 begin-escape .value_required:n = true ,
748 begin-escape .usage:n = preamble ,
749
750 end-escape .code:n =
751 \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
752 end-escape .value_required:n = true ,
753 end-escape .usage:n = preamble ,
754
755 begin-escape-math .code:n =
756 \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
757 begin-escape-math .value_required:n = true ,
758 begin-escape-math .usage:n = preamble ,
759
760 end-escape-math .code:n =
761 \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
762 end-escape-math .value_required:n = true ,
763 end-escape-math .usage:n = preamble ,
764
765 comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
766 comment-latex .value_required:n = true ,
767 comment-latex .usage:n = preamble ,
768
769 label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
770 label-as-zlabel .usage:n = preamble ,
771
772 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
773 math-comments .usage:n = preamble ,

```

Now, general keys.

```

774 language .code:n =
775 \str_set:N \l_piton_language_str { \str_lowercase:n { #1 } } ,
776 language .value_required:n = true ,
777 path .code:n =
778 \seq_clear:N \l_@@_path_seq
779 \clist_map_inline:nn { #1 }
780 {
781 \str_set:Nn \l_tmpa_str { ##1 }
782 \seq_put_right:No \l_@@_path_seq { \l_tmpa_str }
783 } ,
784 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.

```

785 path .initial:n = . ,
786 path-write .str_set:N = \l_@@_path_write_str ,
787 path-write .value_required:n = true ,
788 font-command .tl_set:N = \l_@@_font_command_tl ,
789 font-command .initial:n = \ttfamily ,
790 font-command .value_required:n = true ,
791 font-command+ .code:n
792 = { \tl_put_right:Nn \l_@@_font_command_tl { #1 } } ,

```

```

793 font-command+ .value_required:n = true ,
794 font-command~+ .meta:n = { font-command+ = #1 } ,
795 font-command~+ .value_required:n = true ,
796 gobble .int_set:N = \l_@@_gobble_int ,
797 gobble .default:n = -1 ,
798 auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
799 auto-gobble .value_forbidden:n = true ,
800 env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
801 env-gobble .value_forbidden:n = true ,
802 tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
803 tabs-auto-gobble .value_forbidden:n = true ,
804
805 splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
806
807 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.

```

808 split-separation .tl_set:N = \l_@@_split_separation_tl ,
809 split-separation .value_required:n = true ,
810 split-separation .initial:n = \vspace { \baselineskip } \vspace { -1.25pt } ,
811
812 split-separation+ .code:n =
813   \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
814 split-separation+ .value_required:n = true ,
815 split-separation~+ .meta:n = { split-separation+ = #1 } ,
816 add-to-split-separation .meta:n = { split-separation+ = #1 } ,
817
818 marker .code:n =
819   \bool_lazy_or:nnTF
820     \l_@@_in_PitonInputFile_bool
821     \l_@@_in_PitonOptions_bool
822     { \keys_set:nn { PitonOptions / marker } { #1 } }
823     { \@@_error:n { Invalid-key } } ,
824 marker .value_required:n = true ,
825
826 line-numbers .code:n =
827   \keys_set:nn { PitonOptions / line-numbers } { #1 } ,

```

The following line is mandatory.

```

828 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).

```

829 splittable .int_set:N = \l_@@_splittable_int ,
830 splittable .default:n = 1 ,
831 splittable .initial:n = 100 ,
832 background-color .code:n =
833   \clist_set:Nn \l_@@_bg_color_clist { #1 }

```

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

```

834   \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
835 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`).

```

836 prompt-background-color .tl_set:N = \l_@@_prompt_bg_color_tl ,
837 prompt-background-color .value_required:n = true ,
838 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.

```

839 print .bool_set:N = \l_@@_print_bool ,
840 print .value_required:n = true ,
841 print .initial:n = true ,
842
843 width .code:n =
844   \str_if_eq:nnTF { #1 } { min }
845   {
846     \bool_set_true:N \l_@@_minimize_width_bool
847     \dim_zero:N \l_@@_width_dim
848   }
849   {
850     \bool_set_false:N \l_@@_minimize_width_bool
851     \dim_set:Nn \l_@@_width_dim { #1 }
852   } ,
853 width .value_required:n = true ,
854
855 max-width .code:n =
856   \bool_set_true:N \l_@@_minimize_width_bool
857   \dim_set:Nn \l_@@_width_dim { #1 } ,
858 max-width .value_required:n = true ,
859
860 paperclip .code:n =
861   \bool_set_true:N \l_@@_paperclip_bool
862   \tl_if_novalue:nTF { #1 }
863   { \str_set:Nn \l_@@_paperclip_str { } }
864   { \str_set:Nn \l_@@_paperclip_str { #1 } } ,
865
866 annotation .bool_set:N = \l_@@_annotation_bool ,
867
868 write .str_set:N = \l_@@_write_str ,
869 write .value_required:n = true ,
870 no-write .code:n = \str_set_eq:NN \l_@@_write_str \c_empty_str ,
871 no-write .value_forbidden:n = true ,
872 join .code:n =
873   \str_set:Nn \l_@@_join_str { #1 }
874   \seq_if_in:NnF \g_@@_join_seq { #1 }
875   { \seq_gput_right:No \g_@@_join_seq { #1 } } ,
876 join .value_required:n = true ,
877 join-separation .str_set:N = \l_@@_join_separation_str ,
878 join-separation .value_required:n = true ,
879 no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,
880 no-join .value_forbidden:n = true ,
881
882 left-margin .code:n =
883   \str_if_eq:nnTF { #1 } { auto }
884   {
885     \dim_zero:N \l_@@_left_margin_dim
886     \bool_set_true:N \l_@@_left_margin_auto_bool
887   }
888   {
889     \dim_set:Nn \l_@@_left_margin_dim { #1 }
890     \bool_set_false:N \l_@@_left_margin_auto_bool
891   } ,
892 left-margin .value_required:n = true ,
893
894 right-margin .code:n =
895   \str_if_eq:nnTF { #1 } { auto }
896   {
897     \dim_zero:N \l_@@_right_margin_dim
898     \bool_set_true:N \l_@@_right_margin_auto_bool
899   }
900   {
901     \dim_set:Nn \l_@@_right_margin_dim { #1 }

```

```

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

```

```

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

```



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.

```
1024 \NewDocumentCommand \@@_fake_PitonOptions { }
1025 { \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.

```
1026 \int_new:N \g_@@_visual_line_int
1027 \cs_new_protected:Npn \@@_incr_visual_line:
1028 { \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`.

```
1029 \cs_new_protected:Npn \@@_print_number_left:
1030 {
1031   \hbox_overlap_left:n
1032   {
1033     \@@_actually_print_number:n { \int_to_arabic:n { \g_@@_visual_line_int } }
1034     \skip_horizontal:N \l_@@_numbers_sep_dim
1035   }
1036 }
```

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`.

```
1037 \cs_new_protected:Npn \@@_print_number_right:
1038 {
1039   \hbox_overlap_left:n
1040   {
1041     \@@_actually_print_number:n { \l_tmpa_tl }
1042     \int_compare:nNnT \l_@@_bg_colors_int > 0
1043     { \skip_horizontal:n { 0.1 em } }
1044   }
1045 }
```

`\@@_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:`

```
1046 \cs_new_protected:Npn \@@_actually_print_number:n #1
1047 {
1048   \group_begin:
1049   \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`.

```
1050 { \l_@@_line_numbers_format_tl { \@@_draw_number:e { #1 } } }
1051 {
```

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

```
1052   \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
1053   \l_@@_line_numbers_format_tl { #1 }
1054   \pdfextension literal { EMC }
1055 }
1056 \group_end:
1057 }
```

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

```

1058 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
1059 {
1060   \tl_if_novalue:nTF { #3 }

```

The last argument is provided by curryfication.

```

1061   { \@@_NewPitonLanguage:nnn { #1 } { #2 } }

```

The two last arguments are provided by curryfication.

```

1062   { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
1063 }

```

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.

```

1064 \prop_new:N \g_@@_languages_prop

```

```

1065 \keys_define:nn { NewPitonLanguage }
1066 {
1067   morekeywords .code:n = ,
1068   otherkeywords .code:n = ,
1069   sensitive .code:n = ,
1070   keywordsprefix .code:n = ,
1071   moretexcs .code:n = ,
1072   morestring .code:n = ,
1073   morecomment .code:n = ,
1074   moredelim .code:n = ,
1075   moredirectives .code:n = ,
1076   tag .code:n = ,
1077   alsodigit .code:n = ,
1078   alsoletter .code:n = ,
1079   alsoother .code:n = ,
1080   unknown .code:n = \@@_error:n { Unknown-key-NewPitonLanguage }
1081 }

```

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

```

1082 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
1083 {

```

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}{...}`.

```

1084   \tl_set:Nx \l_tmpa_tl
1085   {
1086     \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
1087     \str_lowercase:n { #2 }
1088   }

```

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

```

1089   \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.

```

1090   \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).

```

1091   \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
1092 }
1093 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
1094 {
1095   \hook_gput_code:nnn { begindocument } { . }
1096   { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }

```

```

1097 }
1098 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }

```

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

```

1099 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnnn #1 #2 #3 #4 #5
1100 {

```

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}{...}`

```

1101   \tl_set:Nx \l_tmpa_tl
1102   {
1103     \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
1104     \str_lowercase:n { #4 }
1105   }

```

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`.

```

1106   \prop_get:NnNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl

```

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

```

1107     { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
1108     { \@@_error:n { Language~not~defined } }
1109   }

```

```

1110 \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.

```

1111   { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
1112 \cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }

```

```

1113 \NewDocumentCommand { \piton } { }
1114 { \peek_meaning:NNTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }
1115 \NewDocumentCommand { \@@_piton_standard } { m }
1116 {
1117   \group_begin:
1118   \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
1119   {

```

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.

```

1120     \bool_lazy_or:nnT
1121     \l_@@_break_lines_in_piton_bool
1122     \l_@@_break_strings_anywhere_bool
1123     { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
1124   }

```

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

```

1125   \automatichyphenmode = 1

```

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

```

1126   \cs_set_eq:NN \ \ \c_backslash_str
1127   \cs_set_eq:NN \% \c_percent_str
1128   \cs_set_eq:NN \{ \c_left_brace_str
1129   \cs_set_eq:NN \} \c_right_brace_str
1130   \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.

```

1131   \cs_set_eq:cN { ~ } \space
1132   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1133 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1134 \tl_set:Nx \l_tmpa_tl
1135 {
1136   \lua_now:e
1137   { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1138   { #1 }
1139 }
1140 \bool_if:NTF \l_@@_show_spaces_bool
1141 { \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1142 {
1143   \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.

```

1144   { \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space }
1145 }

```

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

```

1146 \if_mode_math:
1147 \text { \l_@@_font_command_tl \l_tmpa_tl }
1148 \else:
1149   \l_@@_font_command_tl \l_tmpa_tl
1150 \fi:
1151 \group_end:
1152 }

```

```

1153 \NewDocumentCommand { \@@_piton_verbatim } { v }
1154 {
1155   \group_begin:
1156   \automatichyphenmode = 1
1157   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1158 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1159 \tl_set:Nx \l_tmpa_tl
1160 {
1161   \lua_now:e
1162   { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1163   { #1 }
1164 }
1165 \bool_if:NTF \l_@@_show_spaces_bool
1166 { \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1167 \if_mode_math:
1168 \text { \l_@@_font_command_tl \l_tmpa_tl }
1169 \else:
1170   \l_@@_font_command_tl \l_tmpa_tl
1171 \fi:
1172 \group_end:
1173 }

```

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).

```

1174 \cs_new_protected:Npn \@@_piton:n #1
1175 { \tl_if_blank:NF { #1 } { \@@_piton_i:n { #1 } } }
1176
1177 \cs_new_protected:Npn \@@_piton_i:n #1
1178 {
1179   \group_begin:
1180   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

```

1181 \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1182 \cs_set:cpn { pitonStyle _ Prompt } { }
1183 \cs_set_eq:NN \@@_leading_space: \space
1184 \cs_set_eq:NN \@@_trailing_space: \space
1185 \tl_set:Ne \l_tmpa_tl
1186 {
1187   \lua_now:e
1188   { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1189   { #1 }
1190 }
1191 \bool_if:NT \l_@@_show_spaces_bool
1192 { \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { _ } } % U+2423
1193 \@@_replace_spaces:o \l_tmpa_tl
1194 \group_end:
1195 }

```

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

```

1196 \cs_new_protected:Npn \@@_pre_composition:
1197 {
1198   \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
1199   {
1200     \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).

```

1201     \str_if_empty:NF \l_@@_box_str
1202     { \bool_set_true:N \l_@@_minimize_width_bool }
1203   }

```

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).

```

1204   \dim_set:Nn \l_@@_listing_width_dim
1205   {
1206     \bool_if:NTF \l_@@_tcolorbox_bool
1207     {
1208       \l_@@_width_dim -
1209       ( \kvtcb@left@rule
1210       + \kvtcb@leftupper
1211       + \kvtcb@boxsep * 2
1212       + \kvtcb@rightupper
1213       + \kvtcb@right@rule )
1214     }
1215     { \l_@@_width_dim }
1216   }
1217   \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1218   \automatichyphenmode = 1
1219   \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1220   \g_@@_def_vertical_commands_tl
1221   \int_gzero:N \g_@@_line_int
1222   \int_gzero:N \g_@@_nb_lines_int
1223   \dim_zero:N \parindent
1224   \dim_zero:N \lineskip
1225   \dim_zero:N \parskip
1226
1227   \seq_gclear:N \g_@@_visual_line_numbers_seq
1228
1229   \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

For efficiency, we keep in \l\_@@\_bg\_colors\_int the length of \l\_@@\_bg\_color\_clist.

```

1230   \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1231   { \bool_set_true:N \l_@@_bg_bool }

```

```

1232 \bool_gset_false:N \g_@@_rowcolor_inside_bool
1233 \IfPackageLoadedTF { zref-base }
1234 {
1235     \bool_if:NTF \g_@@_label_as_zlabel_bool
1236     { \cs_set_eq:NN \label \@@_zlabel:n }
1237     { \cs_set_eq:NN \label \@@_label:n }
1238     \cs_set_eq:NN \zlabel \@@_zlabel:n
1239 }
1240 { \cs_set_eq:NN \label \@@_label:n }
1241 \l_@@_font_command_tl
1242 }

```

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`.

```

1243 \cs_new_protected:Npn \@@_compute_margin:N #1
1244 {
1245     \use:e
1246     {
1247         \bool_if:NTF \l_@@_skip_empty_lines_bool
1248         { \lua_now:n { piton.CountNonEmptyLines(token.scan_argument()) } }
1249         { \lua_now:n { piton.CountLines(token.scan_argument()) } }
1250         { \l_@@_listing_tl }
1251     }
1252     \hbox_set:Nn \l_tmpa_box
1253     {
1254         \l_@@_line_numbers_format_tl
1255         \int_to_arabic:n
1256         {
1257             \g_@@_visual_line_int
1258             +
1259             \bool_if:NTF \l_@@_skip_empty_lines_bool
1260             { \l_@@_nb_non_empty_lines_int }
1261             { \g_@@_nb_lines_int }
1262         }
1263     }
1264     \dim_set:Nn #1 { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1265 }

```

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.

```

1266 \cs_new_protected:Npn \@@_compute_code_width:
1267 {
1268     \dim_set:Nn \l_@@_code_width_dim
1269     {
1270         \l_@@_listing_width_dim
1271         -
1272         (
1273             \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1274             {
1275                 \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1276                 { \l_@@_left_margin_dim }
1277                 { 0.5 em }
1278             +

```

```

1279         \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1280         { \l_@@_right_margin_dim }
1281         { 0.5 em }
1282     }
1283     { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1284 )
1285 }
1286 }

```

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:`.

```

1287 \cs_new_protected:Npn \@@_recompute_listing_width:
1288 {
1289     \dim_set:Nn \l_@@_listing_width_dim
1290     {
1291         \box_wd:N \g_@@_output_box
1292         +
1293         \int_compare:nNnTF \l_@@_bg_colors_int > \c_zero_int
1294         {
1295             \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1296             { \l_@@_left_margin_dim }
1297             { 0.5 em }
1298         }
1299         +
1300         \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1301         { \l_@@_right_margin_dim }
1302         { 0.5 em }
1303     }
1304     { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1305 }

```

```

1306 \cs_new_protected:Npn \@@_store_body:n #1
1307 {

```

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

```

1308     \tl_set:Nc \obeyedline { \char_generate:nn { 13 } { 11 } }
1309     \tl_set:Nc \l_@@_listing_tl { #1 }
1310     \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1311 }

```

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

```

1312 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nnnnn
1313 {
1314     \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1315     {
1316         \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1317         #4
1318         \@@_pre_composition:
1319         \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1320         {
1321             \int_gset:Nn \g_@@_visual_line_int
1322             { \l_@@_number_lines_start_int - 1 }
1323         }
1324         \bool_if:NT \g_@@_beamer_bool
1325         { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1326         \bool_if:NT \g_@@_footnote_bool \savenotes
1327         \@@_composition:
1328         \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1329         { \@@_create_paperclip_annotation: }
1330         \bool_if:NT \g_@@_footnote_bool \endsavenotes

```

```

1331     #5
1332   }
1333   { \ignorespacesafterend }
1334 }

```

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

```

1335 \cs_new_protected:Npn \@@_create_paperclip_annotation:
1336 {
1337   \marginalia
1338   {
1339     \vspace* { - 0.8 em }
1340     \hbox:n
1341     {
1342       \vrule-height~0~pt~depth~12~pt~width~0~pt
1343       \bool_if:NT \l_@@_annotation_bool
1344       {
1345         \lua_now:n
1346         {

```

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!

```

1347         pdf.immediateobj
1348         ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1349       }
1350     \pdfextension annot~width~5pt~height~10pt~depth~0pt
1351     {
1352       /Subtype /Text
1353       /Contents~\pdf_object_ref_last:
1354       /Name /Note
1355       /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).

```

1356       /ReplyType /Group

```

Adds the bit 10 which means LockedContents.

```

1357       /F~512
1358       /C [0.8~0.8~0.8]
1359     }
1360     \hspace* { 7 mm }
1361   }
1362   \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1363 }
1364 }
1365 }

```

```

1366 \cs_new_protected:Npn \@@_create_paperclip:
1367 {
1368   \str_if_empty:NT \l_@@_paperclip_str
1369   {
1370     \int_gincr:N \g_@@_paperclip_int
1371     \str_set:Ne \l_@@_paperclip_str { listing\_int_use:N \g_@@_paperclip_int .txt }
1372   }

```

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

```

1373   \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1374   \box_move_down:nn
1375   { 10 pt }
1376   {
1377     \hbox:n
1378     {
1379       \pdfextension annot~width~10pt~height~20pt~depth~0pt
1380       {
1381         /Subtype /FileAttachment

```



```

1382             /Name /Paperclip
1383             /F-8 % no zoom
/Contents will be used as info-bulle and description of the file in the panel of the embedded files.
1384             /Contents (The~computer~listing)
1385             /FS <<
1386                 /Type /Filespec
1387                 /F (\l_@@_paperclip_str)
1388                 /EF << /F~\pdf_object_ref_last: >>
1389                 /AFRelationship /Supplement
1390             >>
1391         }
1392     }
1393 }
1394 }

```

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

```

1395 \NewDocumentCommand { \NewPitonEnvironment } { }
1396 { \@@_DefinePitonEnvironment:nnnnn { New } }
1397 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1398 { \@@_DefinePitonEnvironment:nnnnn { Declare } }
1399 \NewDocumentCommand { \RenewPitonEnvironment } { }
1400 { \@@_DefinePitonEnvironment:nnnnn { Renew } }
1401 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1402 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

1403 \cs_new_protected:Npn \@@_translate_beamer_env:n
1404 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1405 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1406 \cs_new_protected:Npn \@@_composition:
1407 {
1408     \str_if_empty:NT \l_@@_box_str
1409     {
1410         \mode_if_vertical:F
1411         { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1412     }
1413     \bool_if:NT \l_@@_line_numbers_bool
1414     {
1415         \bool_lazy_and:nnT
1416         { \l_@@_left_margin_auto_bool }
1417         { \str_if_eq_p:ee \l_@@_line_numbers_position_str { left } }
1418         { \@@_compute_margin:N \l_@@_left_margin_dim }
1419         \bool_lazy_and:nnT
1420         { \l_@@_right_margin_auto_bool }
1421         { \str_if_eq_p:ee \l_@@_line_numbers_position_str { right } }
1422         { \@@_compute_margin:N \l_@@_right_margin_dim }
1423     }
1424     \lua_now:e
1425     {
1426         piton.join_separation = "\l_@@_join_separation_str"
1427         piton.join = "\l_@@_join_str"
1428         piton.write = "\l_@@_write_str"
1429         piton.path_write = "\l_@@_path_write_str"
1430     }
1431     \noindent
1432     \bool_if:NTF \l_@@_print_bool
1433     {

```

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.

```

1434     \bool_if:NTF \l_@@_split_on_empty_lines_bool
1435     { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1436     {
1437         \@@_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).

```

1438     \bool_if:NTF \l_@@_tcolorbox_bool
1439     {
1440         \str_if_empty:NTF \l_@@_box_str
1441         \@@_composition_iii:
1442         \@@_composition_iv:
1443     }
1444     {
1445         \str_if_empty:NTF \l_@@_box_str
1446         \@@_composition_i:
1447         \@@_composition_ii:
1448     }
1449 }
1450 }
1451 { \@@_gobble_parse_no_print:o \l_@@_listing_tl }
1452 }

```

`\@@_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=...`

```

1453 \cs_new_protected:Npn \@@_composition_i:
1454 {

```

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.

```

1455     \box_clear:N \g_tmpa_box

```

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

```

1456     \box_clear_new:N \g_@@_line_box

```

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

```

1457     \vbox_set:Nn \l_tmpa_box
1458     {
1459         \vbox_unpack_drop:N \g_@@_output_box
1460         \bool_gset_false:N \g_tmpa_bool
1461         \unskip \unskip
1462         \bool_gset_false:N \g_tmpa_bool
1463         \bool_do_until:nn \g_tmpa_bool
1464         {
1465             \unskip \unskip \unskip
1466             \unpenalty \unkern
1467             \box_set_to_last:N \l_@@_line_box
1468             \box_if_empty:NTF \l_@@_line_box
1469             { \bool_gset_true:N \g_tmpa_bool }
1470             {
1471                 \vbox_gset:Nn \g_tmpa_box
1472                 {
1473                     \vbox_unpack:N \g_tmpa_box
1474                     \box_use:N \l_@@_line_box
1475                 }
1476             }
1477         }
1478     }

```

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

```

1479     \bool_gset_false:N \g_tmpa_bool

```

```

1480 \int_zero:N \g_@@_line_int
1481 \bool_do_until:nn \g_tmpa_bool
1482 {

```

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`.

```

1483 \vbox_gset:Nn \g_tmpa_box
1484 {
1485     \vbox_unpack_drop:N \g_tmpa_box
1486     \box_gset_to_last:N \g_@@_line_box
1487 }

```

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.

```

1488 \box_if_empty:NTF \g_@@_line_box
1489 { \bool_gset_true:N \g_tmpa_bool }
1490 {
1491     \box_use:N \g_@@_line_box
1492     \int_gincr:N \g_@@_line_int
1493     \par
1494     \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`.

```

1495     \@@_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.

```

1496     \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int }
1497     { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int } }
1498     \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1499     { \mode_leave_vertical: }
1500 }
1501 }
1502 \skip_vertical:n { 2.5 pt }
1503 }

```

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

```

1504 \cs_new_protected:Npn \@@_composition_ii:
1505 {
1506     \use:e { \begin { minipage } [ \l_@@_box_str ] }
1507     { \l_@@_listing_width_dim }

```

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

```

1508     \vbox_unpack:N \g_@@_output_box

```

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

```

1509     \kern 2.5 pt
1510     \end { minipage }
1511 }

```

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

```

1512 \cs_new_protected:Npn \@@_composition_iii:
1513 {
1514     \use:e
1515     {
1516         \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.

```

1517         [ breakable , text-width = \l_@@_listing_width_dim ]
1518     }
1519     \par
1520     \vbox_unpack:N \g_@@_output_box
1521     \end { tcolorbox }
1522 }

```

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

```

1523 \cs_new_protected:Npn \@@_composition_iv:
1524 {
1525   \use:e
1526   {
1527     \begin { tcolorbox }
1528     [
1529       hbox ,
1530       text~width = \l_@@_listing_width_dim ,
1531       nobeforeafter ,
1532       box~align =
1533         \str_case:Nn \l_@@_box_str
1534         {
1535           t { top }
1536           b { bottom }
1537           c { center }
1538           m { center }
1539         }
1540     ]
1541   }
1542   \box_use:N \g_@@_output_box
1543   \end { tcolorbox }
1544 }

```

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.

```

1545 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1546 {
1547   \int_case:nn
1548   {
1549     \lua_now:e
1550     {
1551       tex.sprint
1552         ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1553     }
1554   }
1555   { 1 { \penalty 100 } 2 \nobreak }
1556 }

```

`\@@_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`.

```

1557 \cs_new_protected:Npn \@@_create_output_box:
1558 {
1559   \@@_compute_code_width:
1560   \vbox_gset:Nn \g_@@_output_box
1561     { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1562   \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1563   \bool_lazy_any:nT
1564     {
1565       { \int_compare_p:nNn \l_@@_bg_colors_int > \c_zero_int }
1566       { \g_@@_rowcolor_inside_bool }
1567     }
1568     {
1569       \l_@@_line_numbers_bool
1570       &&
1571       \str_if_eq_p:ee \l_@@_line_numbers_position_str { right }
1572     }
1573   \@@_add_bg_and_right_nb_to_output_box:
1574 }

```

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:`.

```
1575 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_output_box:
1576 {
1577     \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`.

```
1578 \vbox_set:Nn \l_tmpa_box
1579 {
1580     \vbox_unpack_drop:N \g_@@_output_box
```

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

```
1581 \bool_gset_false:N \g_tmpa_bool
1582 \unskip \unskip
```

We begin the loop.

```
1583 \bool_do_until:n \g_tmpa_bool
1584 {
1585     \unskip \unskip \unskip
1586     \int_set_eq:NN \l_tmpa_int \lastpenalty
1587     \unpenalty \unkern
```

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

```
1588 \box_set_to_last:N \l_@@_line_box
1589 \box_if_empty:NTF \l_@@_line_box
1590 { \bool_gset_true:N \g_tmpa_bool }
1591 { }
```

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

```
1592 \vbox_gset:Nn \g_@@_output_box
1593 {
```

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`.

```
1594     \@@_add_bg_and_right_nb_to_line_and_use:
1595     \kern -2.5 pt
1596     \penalty \l_tmpa_int
1597     \vbox_unpack:N \g_@@_output_box
1598 }
1599 }
1600 \int_gdecr:N \g_@@_line_int
1601 }
1602 }
1603 }
```

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

```
1604 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1605 {
1606     \lua_now:e
1607     {
1608         piton.GobbleParseNoPrint
1609         (
1610             '\l_piton_language_str' ,
1611             \int_use:N \l_@@_gobble_int ,
1612             token.scan_argument ( )
1613         )
1614     }
1615 }
1616 \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.

```

1617 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1618 {
1619   \lua_now:e
1620   {
1621     piton.RetrieveGobbleParse
1622     (
1623       '\l_piton_language_str' ,
1624       \int_use:N \l_@@_gobble_int ,
1625       \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1626       { \int_eval:n { - \l_@@_splittable_int } }
1627       { \int_use:N \l_@@_splittable_int } ,
1628       token.scan_argument ( )
1629     )
1630   }
1631 }
1632 \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.

```

1633 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1634 {
1635   \lua_now:e
1636   {
1637     piton.RetrieveGobbleSplitParse
1638     (
1639       '\l_piton_language_str' ,
1640       \int_use:N \l_@@_gobble_int ,
1641       \int_use:N \l_@@_splittable_int ,
1642       token.scan_argument ( )
1643     )
1644   }
1645 }
1646 \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`.

```

1647 \bool_if:NTF \g_@@_beamer_bool
1648 {
1649   \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1650   {
1651     \keys_set:nn { PitonOptions } { #2 }
1652     \begin { actionenv } < #1 >
1653   }
1654   { \end { actionenv } }
1655 }
1656 {
1657   \NewPitonEnvironment { Piton } { 0 { } }
1658   { \keys_set:nn { PitonOptions } { #1 } }
1659   { }
1660 }

1661 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1662 {
1663   \mode_if_vertical:F { \par }
1664   \group_begin:
1665   \seq_concat:NNN
1666     \l_file_search_path_seq
1667     \l_@@_path_seq

```

```

1668 \l_file_search_path_seq
1669 \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1670 {
1671   \@@_input_file:nn { #1 } { #2 }
1672   #4
1673 }
1674 { #5 }
1675 \group_end:
1676 }

1677 \cs_new_protected:Npn \@@_unknown_file:n #1
1678 { \msg_error:nnn { piton } { Unknown~file } { #1 } }

1679 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1680 {
1681   \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1682   {

```

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

```

1683 \iow_log:n { No~file~#3 }
1684 \@@_unknown_file:n { #3 }
1685 }
1686 }
1687 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1688 {
1689   \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1690   {

```

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

```

1691 \iow_log:n { No~file~#3 }
1692 \@@_unknown_file:n { #3 }
1693 }
1694 }
1695 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1696 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```

1697 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1698 {

```

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 >).

```

1699 \tl_if_novalue:nF { #1 }
1700 {
1701   \bool_if:NTF \g_@@_beamer_bool
1702   { \begin { uncoverenv } < #1 > }
1703   { \@@_error_or_warning:n { overlay~without~beamer } }
1704 }
1705 \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.

```

1706 \iow_log:e { (\l_@@_file_name_str) }
1707 \int_zero_new:N \l_@@_first_line_int
1708 \int_zero_new:N \l_@@_last_line_int
1709 \int_set_eq:NN \l_@@_last_line_int \c_max_int
1710 \bool_set_true:N \l_@@_in_PitonInputFile_bool
1711 \keys_set:nn { PitonOptions } { #2 }
1712 \bool_if:NT \l_@@_line_numbers_absolute_bool
1713 { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1714 \bool_if:nTF
1715 {
1716   (
1717     \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1718     || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1719   )
1720   && ! \str_if_empty_p:N \l_@@_begin_range_str

```

```

1721     }
1722     {
1723         \@@_error_or_warning:n { bad-range-specification }
1724         \int_zero:N \l_@@_first_line_int
1725         \int_set_eq:NN \l_@@_last_line_int \c_max_int
1726     }
1727     {
1728         \str_if_empty:NF \l_@@_begin_range_str
1729         {
1730             \@@_compute_range:
1731             \bool_lazy_or:nnT
1732             \l_@@_marker_include_lines_bool
1733             { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1734             {
1735                 \int_decr:N \l_@@_first_line_int
1736                 \int_incr:N \l_@@_last_line_int
1737             }
1738         }
1739     }
1740     \@@_pre_composition:
1741     \bool_if:NT \l_@@_line_numbers_absolute_bool
1742     { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1743     \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1744     {
1745         \int_gset:Nn \g_@@_visual_line_int
1746         { \l_@@_number_lines_start_int - 1 }
1747     }

```

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

```

1748     \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1749     { \int_gzero:N \g_@@_visual_line_int }
1750     \lua_now:e
1751     {

```

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

```

1752         piton.ReadFile(
1753             '\l_@@_file_name_str' ,
1754             \int_use:N \l_@@_first_line_int ,
1755             \int_use:N \l_@@_last_line_int )
1756     }
1757     \@@_composition:
1758     \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.

```

1759     \tl_if_novalue:nF { #1 }
1760     { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1761 }

```

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

```

1762 \cs_new_protected:Npn \@@_compute_range:
1763 {

```

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

```

1764     \str_set:Nne \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1765     \str_set:Nne \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`.

```

1766     \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1767     \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1768     \lua_now:e

```



```

1769     {
1770         piton.ComputeRange
1771         ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1772     }
1773 }

```

## 2.8 The styles

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

```

1774 \NewDocumentCommand { \PitonStyle } { m }
1775 {
1776     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1777     { \use:c { pitonStyle _ #1 } }
1778 }

```

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”.

```

1779 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1780 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1781 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1782 {
1783     \str_clear_new:N \l_@@_SetPitonStyle_option_str
1784     \str_set:Nx \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1785     \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1786     { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1787     \keys_set:nn { piton / Styles } { #2 }
1788 }

```

```

1789 \cs_new_protected:Npn \@@_math_scantokens:n #1
1790 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

```

```

1791 \clist_new:N \g_@@_styles_clist
1792 \clist_gset:Nn \g_@@_styles_clist
1793 {
1794     Comment ,
1795     Comment.Internal ,
1796     Comment.LaTeX ,
1797     Discard ,
1798     Delim ,
1799     Exception ,
1800     FormattingType ,
1801     Identifier.Internal ,
1802     Identifier ,
1803     InitialValues ,
1804     Interpol.Inside ,
1805     Keyword ,
1806     Keyword.Governing ,
1807     Keyword.Constant ,
1808     Keyword2 ,
1809     Keyword3 ,
1810     Keyword4 ,
1811     Keyword5 ,
1812     Keyword6 ,
1813     Keyword7 ,
1814     Keyword8 ,
1815     Keyword9 ,
1816     Name.Builtin ,
1817     Name.Class ,
1818     Name.Constructor ,

```

```

1819 Name.Decorator ,
1820 Name.Field ,
1821 Name.Function ,
1822 Name.Module ,
1823 Name.Namespace ,
1824 Name.Table ,
1825 Name.Type ,
1826 Number ,
1827 Number.Internal ,
1828 Operator ,
1829 Operator.Word ,
1830 Preproc ,
1831 Prompt ,
1832 Punct ,
1833 String.Doc ,
1834 String.Doc.Internal ,
1835 String.Interpol ,
1836 String.Long ,
1837 String.Long.Internal ,
1838 String.Short ,
1839 String.Short.Internal ,
1840 Tag ,
1841 TypeParameter ,
1842 UserFunction ,

```

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

```

1843 TypeExpression ,

```

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

```

1844 Directive
1845 }

1846 \clist_map_inline:Nn \g_@@_styles_clist
1847 {
1848   \keys_define:nn { piton / Styles }
1849   {
1850     #1 .value_required:n = true ,
1851     #1 .code:n =
1852       \tl_set:cn
1853       {
1854         pitonStyle _
1855         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1856         { \l_@@_SetPitonStyle_option_str _ }
1857         #1
1858       }
1859     { ##1 }
1860   }
1861 }

1862
1863 \keys_define:nn { piton / Styles }
1864 {
1865   String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1866   String      .value_required:n = true ,
1867   Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1868   Comment.Math .value_required:n = true ,
1869   unknown     .code:n = \@@_unknown_style:
1870 }

```

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.

```

1871 \cs_new_protected:Npn \@@_unknown_style:
1872 {
1873   \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1874   {

```

```

1875     \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1876     \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`.

```

1877     \bool_lazy_and:nnTF
1878     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1879     {
1880         \str_if_eq_p:Vn \l_tmpa_str { Module }
1881         ||
1882         \str_if_eq_p:Vn \l_tmpa_str { Type }
1883     }

```

Now, we will create a new style.

```

1884     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1885     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1886 }
1887 { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1888 }

```

```

1889 \SetPitonStyle[OCaml]
1890 {
1891     TypeExpression =
1892     {
1893         \SetPitonStyle [ OCaml ]
1894         {
1895             Identifier = \PitonStyle { Name.Type } ,
1896             Name.Builtin = \PitonStyle { Name.Type}
1897         }
1898         \@@_piton:n
1899     }
1900 }

```

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`.

```

1901 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that clist.

```

1902 \clist_gsort:Nn \g_@@_styles_clist
1903 {
1904     \str_compare:nNnTF { #1 } < { #2 }
1905     \sort_return_same:
1906     \sort_return_swapped:
1907 }

```

```

1908 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1909
1910 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1911
1912 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1913 {
1914     \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`.

```

1915     \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space
1916     \seq_clear:N \l_tmpa_seq
1917     \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1918     \seq_use:Nn \l_tmpa_seq { \- }
1919 }

```

```

1920 \cs_new_protected:Npn \@@_comment:n #1
1921   { \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”.

```

1922 \cs_new_protected:Npn \PitonSpaceSubstitute #1 % noqa
1923   {
1924     \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1925       {
1926         \tl_set:Nn \l_tmpa_tl { #1 }
1927         \tl_replace_all:Nvn \l_tmpa_tl
1928           \c_catcode_other_space_tl
1929           \@@_breakable_space:
1930         \l_tmpa_tl
1931       }
1932     { #1 }
1933   }

```

```

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

```

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

```

1941       \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1942         {
1943           \tl_set:Nn \l_tmpa_tl { #1 }
1944           \tl_replace_all:Nvn \l_tmpa_tl
1945             \c_catcode_other_space_tl
1946             \@@_breakable_space:
1947           \l_tmpa_tl
1948         }
1949       { #1 }
1950     }
1951   }
1952 }
1953 \cs_new_protected:Npn \@@_string_short:n #1
1954   {
1955     \PitonStyle { String.Short }
1956     {
1957       \bool_if:NT \l_@@_break_strings_anywhere_bool
1958         { \@@_actually_break_anywhere:n }
1959       { #1 }
1960     }
1961   }
1962 \cs_new_protected:Npn \@@_string_doc:n #1
1963   {
1964     \PitonStyle { String.Doc }
1965     {
1966       \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1967         {
1968           \tl_set:Nn \l_tmpa_tl { #1 }
1969           \tl_replace_all:Nvn \l_tmpa_tl

```

```

1970         \c_catcode_other_space_tl
1971         \@@_breakable_space:
1972         \l_tmpa_tl
1973     }
1974     { #1 }
1975 }
1976 }
1977 \cs_new_protected:Npn \@@_number:n #1
1978 {
1979     \PitonStyle { Number }
1980     {
1981         \bool_if:NT \l_@@_break_numbers_anywhere_bool
1982         { \@@_actually_break_anywhere:n }
1983         { #1 }
1984     }
1985 }

```

## 2.9 The initial styles

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

```

1986 \SetPitonStyle
1987 {
1988     Comment           = \color [ HTML ] { 0099FF } \itshape ,
1989     Comment.Internal  = \@@_comment:n ,
1990     Exception         = \color [ HTML ] { CC0000 } ,
1991     Keyword           = \color [ HTML ] { 006699 } \bfseries ,
1992     Keyword.Governing = \color [ HTML ] { 006699 } \bfseries ,
1993     Keyword.Constant  = \color [ HTML ] { 006699 } \bfseries ,
1994     Name.Builtin      = \color [ HTML ] { 336666 } ,
1995     Name.Decorator     = \color [ HTML ] { 9999FF } ,
1996     Name.Class        = \color [ HTML ] { 00AA88 } \bfseries ,
1997     Name.Function     = \color [ HTML ] { CC00FF } ,
1998     Name.Namespace    = \color [ HTML ] { 00CCFF } ,
1999     Name.Constructor  = \color [ HTML ] { 006000 } \bfseries ,
2000     Name.Field        = \color [ HTML ] { AA6600 } ,
2001     Name.Module       = \color [ HTML ] { 0060A0 } \bfseries ,
2002     Name.Table        = \color [ HTML ] { 309030 } ,
2003     Number            = \color [ HTML ] { FF6600 } ,
2004     Number.Internal   = \@@_number:n ,
2005     Operator          = \color [ HTML ] { 555555 } ,
2006     Operator.Word     = \bfseries ,
2007     String            = \color [ HTML ] { CC3300 } ,
2008     String.Long.Internal = \@@_string_long:n ,
2009     String.Short.Internal = \@@_string_short:n ,
2010     String.Doc.Internal = \@@_string_doc:n ,
2011     String.Doc        = \color [ HTML ] { CC3300 } \itshape ,
2012     String.Interpol    = \color [ HTML ] { AA0000 } ,
2013     Comment.LaTeX     = \normalfont \color [ rgb ] { .468, .532, .6 } ,
2014     Name.Type         = \color [ HTML ] { 336666 } ,
2015     InitialValues     = \@@_piton:n ,
2016     Interpol. Inside  = { \l_@@_font_command_tl \@@_piton:n } ,
2017     TypeParameter     = \color [ HTML ] { 336666 } \itshape ,
2018     Preproc           = \color [ HTML ] { AA6600 } \slshape ,

```

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

```

2019     Identifier.Internal = \@@_identifier:n ,
2020     Identifier          = ,
2021     Directive          = \color [ HTML ] { AA6600 } ,
2022     Tag                = \colorbox { gray!10 } ,

```

```

2023 UserFunction      = \PitonStyle { Identifier } ,
2024 Prompt            = ,
2025 Discard            = \use_none:n
2026 }

```

## 2.10 Styles specific to the language expl

```

2027 \clist_new:N \g_@@_expl_styles_clist
2028 \clist_gset:Nn \g_@@_expl_styles_clist
2029 {
2030   Scope.l ,
2031   Scope.g ,
2032   Scope.c
2033 }

2034 \clist_map_inline:Nn \g_@@_expl_styles_clist
2035 {
2036   \keys_define:nn { piton / Styles }
2037   {
2038     #1 .value_required:n = true ,
2039     #1 .code:n =
2040       \tl_set:cn
2041       {
2042         pitonStyle _
2043         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
2044         { \l_@@_SetPitonStyle_option_str _ }
2045         #1
2046       }
2047     { ##1 }
2048   }
2049 }

2050 \SetPitonStyle [ expl ]
2051 {
2052   Scope.l      = ,
2053   Scope.g      = \bfseries ,
2054   Scope.c      = \slshape ,
2055   Type.bool    = \color [ HTML ] { AA6600 } ,
2056   Type.box     = \color [ HTML ] { 267910 } ,
2057   Type.clist   = \color [ HTML ] { 309030 } ,
2058   Type.fp      = \color [ HTML ] { FF3300 } ,
2059   Type.int     = \color [ HTML ] { FF6600 } ,
2060   Type.seq     = \color [ HTML ] { 309030 } ,
2061   Type.skip    = \color [ HTML ] { OCC060 } ,
2062   Type.str     = \color [ HTML ] { CC3300 } ,
2063   Type.tl      = \color [ HTML ] { AA2200 } ,
2064   Module.bool  = \color [ HTML ] { AA6600 } ,
2065   Module.box   = \color [ HTML ] { 267910 } ,
2066   Module.cs    = \bfseries \color [ HTML ] { 006699 } ,
2067   Module.exp   = \bfseries \color [ HTML ] { 404040 } ,
2068   Module.hbox  = \color [ HTML ] { 267910 } ,
2069   Module.prg   = \bfseries ,
2070   Module.clist = \color [ HTML ] { 309030 } ,
2071   Module.fp    = \color [ HTML ] { FF3300 } ,
2072   Module.int   = \color [ HTML ] { FF6600 } ,
2073   Module.seq   = \color [ HTML ] { 309030 } ,
2074   Module.skip  = \color [ HTML ] { OCC060 } ,
2075   Module.str   = \color [ HTML ] { CC3300 } ,
2076   Module.tl    = \color [ HTML ] { AA2200 } ,
2077   Module.vbox  = \color [ HTML ] { 267910 }
2078 }

```

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

style `Comment.Math` which should be considered only at an “internal style”. However, maybe we will document in a future version the possibility to write change the style *locally* in a document)].

```

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

```

## 2.11 Highlighting some identifiers

```

2084 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
2085 {
2086     \clist_set:Nn \l_tmpa_clist { #2 }
2087     \tl_if_novalue:nTF { #1 }
2088     {
2089         \clist_map_inline:Nn \l_tmpa_clist
2090         { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
2091     }
2092     {
2093         \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
2094         \str_if_eq:onT \l_tmpa_str { current-language }
2095         { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
2096         \clist_map_inline:Nn \l_tmpa_clist
2097         { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
2098     }
2099 }
2100 \cs_new_protected:Npn \@@_identifier:n #1
2101 {
2102     \str_set:Nn \l_tmpa_str { #1 }
2103     \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2104     {
2105         \cs_if_exist_use:cF { PitonIdentifier _ \l_tmpa_str }
2106         { \PitonStyle { Identifier } }
2107     }
2108     { #1 }
2109 }

```

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`).

```

2110 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
2111 {

```

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

```

2112 { \PitonStyle { Name.Function } { #1 } }
2113 \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}`.

```

2114 \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2115 { \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`.**

```

2116 \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
2117 { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
2118 \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.

```

2119 \seq_if_in:NoF \g_@@_languages_seq { \l_piton_language_str }

```

```

2120     { \seq_gput_left:No \g_@@_languages_seq { \l_piton_language_str } }
2121 }

```

```

2122 \NewDocumentCommand \PitonClearUserFunctions { ! o }
2123 {
2124     \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.

```

2125     { \@@_clear_all_functions: }
2126     { \@@_clear_list_functions:n { #1 } }
2127 }

```

```

2128 \cs_new_protected:Npn \@@_clear_list_functions:n #1
2129 {
2130     \clist_set:Nn \l_tmpa_clist { #1 }
2131     \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
2132     \clist_map_inline:nn { #1 }
2133     { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
2134 }

```

```

2135 \cs_new_protected:Npn \@@_clear_functions_i:n #1
2136 { \@@_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).

```

2137 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2138 {
2139     \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2140     {
2141         \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2142         { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }
2143         \seq_gclear:c { g_@@_functions _ #1 _ seq }
2144     }
2145 }
2146 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }

```

```

2147 \cs_new_protected:Npn \@@_clear_functions:n #1
2148 {
2149     \@@_clear_functions_i:n { #1 }
2150     \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2151 }

```

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

```

2152 \cs_new_protected:Npn \@@_clear_all_functions:
2153 {
2154     \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2155     \seq_gclear:N \g_@@_languages_seq
2156 }

```

```

2157 \AtEndDocument
2158 {

```

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

```

2159     \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.

```

2160     \IfPDFManagementActiveTF
2161     { \@@_join_files: }
2162     { \@@_join_files_legacy: }
2163 }

```



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

```

2164 \cs_new_protected:Npn \@@_join_files:
2165 {
2166   \seq_map_inline:Nn \g_@@_join_seq
2167   {
2168     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2169     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2170     \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2171     {
2172       <<
2173         /Type /Filespec
2174         /UF <\l_tmpa_str>
2175         /EF << /F~\pdf_object_ref_last: >>
2176         /Desc (Computer~listing)
2177         /AFRelationship /Supplement
2178       >>
2179     }
2180   }
2181 }

```

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 techniques.

```

2182 \cs_new_protected:Npn \@@_join_files_legacy:
2183 {
2184   \seq_map_inline:Nn \g_@@_join_seq
2185   {
2186     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2187     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2188     \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`.

```

2189   {
2190     /Subtype /FileAttachment
2191     /F~2
2192     /Name /Paperclip
2193     /Contents (Computer~listing)
2194     /FS <<
2195       /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).

```

2196       /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`).

```

2197       /EF << /F~\pdf_object_ref_last: >>
2198       /AFRelationship /Supplement
2199     >>
2200   }
2201 }
2202 }

```

## 2.12 Spaces of indentation

```

2203 \cs_new_protected:Npn \@@_define_leading_space_normal:
2204 {
2205   \cs_set_protected:Npn \@@_leading_space:

```

```

2206     {
2207         \int_gincr:N \g_@@_indentation_int

```

Be careful: the \hbox:n is mandatory.

```

2208         \hbox:n { ~ }
2209     }
2210 }
2211 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2212 {
2213     \cs_set_protected:Npn \@@_leading_space:
2214     {
2215         \int_gincr:N \g_@@_indentation_int
2216         \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2217         {
2218             \color { white }
2219             \transparent { 0 }
2220             . % previously : □ U+2423
2221         }
2222         \pdfextension literal { EMC }
2223     }
2224 }
2225 \@@_define_leading_space_Foxit:

```

## 2.13 Security

```

2226 \AddToHook { env / piton / before }
2227 { \@@_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.

```

2228 \cs_new_protected:Npn \@@_unknown_key:nn #1 #2
2229 {
2230     \str_set_eq:NN \l_tmpa_str \l_keys_key_str
2231     \str_replace_all:Nnn \l_tmpa_str { ~ } { - }
2232     \str_set:Ne \l_tmpa_str { \str_lowercase:f { \l_tmpa_str } }
2233     \bool_set_false:N \l_tmpa_bool
2234     \clist_map_inline:nn { #1 }
2235     {
2236         \keys_if_exist:neT { ##1 } { \l_tmpa_str }
2237         {
2238             \@@_error:n { key~with~normal~form~exists }
2239             \bool_set_true:N \l_tmpa_bool
2240             \clist_map_break:
2241         }
2242     }
2243     \bool_if:NF \l_tmpa_bool { \@@_error:n { #2 } }
2244 }
2245 \@@_msg_new:nn { key~with~normal~form~exists }
2246 {
2247     The-key~'\l_keys_key_str'~does~not~exists.~It~will~be~ignored.\\
2248     Maybe~you~want~to~use~the~key~'\l_tmpa_str'.
2249 }
2250 \@@_msg_new:nn { No~environment~piton }
2251 {
2252     There~is~no~environment~piton!\\
2253     There~is~an~environment~{Piton}~and~a~command~
2254     \token_to_str:N \piton\ but~there~is~no~environment~

```

```

2255     {piton}.
2256   }

2257 \@@_msg_new:nn { rounded-corners-without~Tikz }
2258   {
2259     TikZ~not~used \\
2260     You~can't~use~the~key~'rounded-corners'~because~
2261     you~have~not~loaded~the~package~TikZ.~
2262     If~you~go~on,~your~key~will~be~ignored.~
2263     You~won't~have~similar~error~till~the~end~of~the~document.
2264   }

2265 \@@_msg_new:nn { tcolorbox~not~loaded }
2266   {
2267     tcolorbox~not~loaded \\
2268     You~can't~use~the~key~'tcolorbox'~because~
2269     you~have~not~loaded~the~package~tcolorbox.~
2270     Use~\token_to_str:N \usepackage[breakable]{tcolorbox}.~
2271     If~you~go~on,~that~key~will~be~ignored.
2272   }

2273 \@@_msg_new:nn { library~breakable~not~loaded }
2274   {
2275     breakable~not~loaded \\
2276     You~can't~use~the~key~'tcolorbox'~because~
2277     you~have~not~loaded~the~library~'breakable'~of~tcolorbox'.~
2278     Use~\token_to_str:N \tcbuselibrary{breakable}~in~the~preamble~
2279     of~your~document.~
2280     If~you~go~on,~that~key~will~be~ignored.
2281   }

2282 \@@_msg_new:nn { Language~not~defined }
2283   {
2284     Language~not~defined \\
2285     The~language~'\l_tmpa_tl'~has~not~been~defined~previously.~
2286     If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2287     will~be~ignored.
2288   }

2289 \@@_msg_new:nn { bad~version~of~piton.lua }
2290   {
2291     Bad~number~version~of~'piton.lua'\\
2292     The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2293     version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2294     address~that~issue.
2295   }

2296 \@@_msg_new:nn { Unknown~key~NewPitonLanguage }
2297   {
2298     Unknown~key~for~\token_to_str:N \NewPitonLanguage.\\
2299     The~key~'\l_keys_key_str'~is~unknown.~
2300     This~key~will~be~ignored.
2301   }

2302 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
2303   {
2304     The~style~'\l_keys_key_str'~is~unknown.\\
2305     This~setting~will~be~ignored.~
2306     The~available~styles~are~(in~alphabetic~order):~
2307     \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2308   }

2309 \@@_msg_new:nn { Invalid~key }
2310   {
2311     Wrong~use~of~key.\\
2312     You~can't~use~the~key~'\l_keys_key_str'~here.~
2313     Your~key~will~be~ignored.
2314   }

```

```

2315 \@@_msg_new:nn { Unknown-key-for-line-numbers }
2316 {
2317   Unknown~key. \\
2318   The~key~'line-numbers / \l_keys_key_str'~is~unknown.~
2319   The~available~keys~of~the~family~'line-numbers'~are~(in~
2320   alphabetic~order):~
2321   absolute,~false,~label-empty-lines,~lmono10-drawn,~
2322   ~position,~resume,~skip-empty-lines,~
2323   sep,~start~and~true.~
2324   Your~key~will~be~ignored.
2325 }
2326 \@@_msg_new:nn { Unknown-key-for-marker }
2327 {
2328   Unknown~key. \\
2329   The~key~'marker / \l_keys_key_str'~is~unknown.~
2330   The~available~keys~of~the~family~'marker'~are~(in~
2331   alphabetic~order):~ beginning,~end~and~include-lines.~
2332   Your~key~will~be~ignored.
2333 }
2334 \@@_msg_new:nn { bad-range-specification }
2335 {
2336   Incompatible~keys.\\
2337   You~can't~specify~the~range~of~lines~to~include~by~using~both~
2338   markers~and~explicit~number~of~lines.~
2339   Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2340 }
2341 \cs_new_nopar:Nn \@@_thepage:
2342 {
2343   \thepage
2344   \cs_if_exist:NT \insertframenummer
2345   {
2346     ~(frame~\insertframenummer
2347     \cs_if_exist:NT \beamer@slidenummer { ,~slide~\insertslidenummer }
2348     )
2349   }
2350 }

```

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`.

```

2351 \@@_msg_new:nn { SyntaxError }
2352 {
2353   Syntax~Error~on~page~\@@_thepage:~\\
2354   Your~code~of~the~language~'\l_piton_language_str'~is~not~
2355   syntactically~correct.~
2356   It~won't~be~printed~in~the~PDF~file.
2357 }
2358 \@@_msg_new:nn { FileError }
2359 {
2360   File~Error.\\
2361   It's~not~possible~to~write~on~the~file~'#1' \\
2362   \sys_if_shell_unrestricted:F
2363   { (try~to~compile~with~'lua~latex~--shell-escape').\\ }
2364   If~you~go~on,~nothing~will~be~written~on~that~file.
2365 }
2366 \@@_msg_new:nn { InexistentDirectory }
2367 {
2368   Inexistent~directory.\\
2369   The~directory~'\l_@@_path_write_str'~
2370   given~in~the~key~'path-write'~does~not~exist.~
2371   Nothing~will~be~written~on~'\l_@@_write_str'.
2372 }

```

```

2373 \@@_msg_new:nn { begin-marker-not-found }
2374 {
2375   Marker-not-found.\\
2376   The-range-\l_@@_begin_range_str'-provided-to-the~
2377   command~\token_to_str:N \PitonInputFile\ has-not-been-found.~
2378   The-whole-file-\l_@@_file_name_str'~will-be-inserted.
2379 }
2380 \@@_msg_new:nn { end-marker-not-found }
2381 {
2382   Marker-not-found.\\
2383   The-marker-of-end-of-the-range-\l_@@_end_range_str'~
2384   provided-to-the-command~\token_to_str:N \PitonInputFile\
2385   has-not-been-found.~The-file-\l_@@_file_name_str'~will~
2386   be-inserted-till-the-end.
2387 }
2388 \@@_msg_new:nn { Unknown-file }
2389 {
2390   Unknown-file. \\
2391   The-file-#1'-is-unknown.~
2392   Your-command~\token_to_str:N \PitonInputFile\ will-be-ignored.
2393 }
2394 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2395 {
2396   \bool_if:NF \g_@@_beamer_bool
2397   { \@@_error_or_warning:n { Without-beamer } }
2398 }
2399 \@@_msg_new:nn { Without-beamer }
2400 {
2401   Key-\l_keys_key_str'~without-Beamer.\\
2402   You-should-not-use-the-key-\l_keys_key_str'~since-you~
2403   are-not-in-Beamer.~However,~you-can-go-on.
2404 }
2405 \@@_msg_new:nn { rowcolor-in-detected-commands }
2406 {
2407   'rowcolor'~forbidden-in-'detected-commands'.\\
2408   You-should-put-'rowcolor'~in-'raw-detected-commands'.~
2409   That-key~will~be-ignored.
2410 }
2411 \@@_msg_new:nnn { Unknown-key-for-PitonOptions }
2412 {
2413   Unknown-key. \\
2414   The-key-\l_keys_key_str'~is-unknown~for~\token_to_str:N \PitonOptions.~
2415   It~will~be-ignored.~
2416   For-a-list-of-the-available-keys,~type-H<return>.
2417 }
2418 {
2419   The-available-keys-are~(in-alphabetic-order):~
2420   annotation,~
2421   add-to-split-separation,~
2422   auto-gobble,~
2423   background-color,~
2424   begin-range,~
2425   box,~
2426   break-lines,~
2427   break-lines-in-piton,~
2428   break-lines-in-Piton,~
2429   break-numbers-anywhere,~
2430   break-strings-anywhere,~
2431   continuation-symbol,~
2432   continuation-symbol-on-indentation,~
2433   detected-beamer-commands,~

```

```

2434 detected-beamer-environments,~
2435 detected-commands,~
2436 end-of-broken-line,~
2437 end-range,~
2438 env-gobble,~
2439 env-used-by-split,~
2440 font-command(+),~
2441 gobble,~
2442 indent-broken-lines,~
2443 join,~
2444 label-as-zlabel,~
2445 language,~
2446 left-margin,~
2447 line-numbers/,~
2448 marker/,~
2449 math-comments,~
2450 no-join,~
2451 no-write,~
2452 path,~
2453 path-write,~
2454 print,~
2455 prompt-background-color,~
2456 raw-detected-commands,~
2457 resume,~
2458 right-margin,~
2459 rounded-corners,~
2460 show-spaces,~
2461 show-spaces-in-strings,~
2462 splittable,~
2463 splittable-on-empty-lines,~
2464 split-on-empty-lines,~
2465 split-separation,~
2466 tabs-auto-gobble,~
2467 tab-size,~
2468 tcolorbox,~
2469 varwidth,~
2470 vertical-detected-commands,~
2471 width-and-write.
2472 }

2473 \@@_msg_new:nn { label-with-lines-numbers }
2474 {
2475   You~can't~use~the~command~\token_to_str:N \label\
2476   or~\token_to_str:N \zlabel\
2477   because~the~key~'line-numbers'~
2478   is~not~active.~If~you~go~on,~that~command~will~ignored.
2479 }

2480 \@@_msg_new:nn { overlay~without~beamer }
2481 {
2482   You~can't~use~an~argument~<...>~for~your~command~
2483   \token_to_str:N \PitonInputFile\ because~you~are~not~
2484   in~Beamer.~If~you~go~on,~that~argument~will~be~ignored.
2485 }

2486 \@@_msg_new:nn { label-as-zlabel-needs-zref-package }
2487 {
2488   The~key~'label-as-zlabel'~requires~the~package~'zref'.~
2489   Please~load~the~package~'zref'~before~setting~the~key.
2490 }
2491 \hook_gput_code:nnn { begindocument } { . }

```

```

2492 {
2493   \bool_if:NT \g_@@_label_as_zlabel_bool
2494   {
2495     \IfPackageLoadedF { zref-base }
2496     { \@@_fatal:n { label-as-zlabel-needs-zref-package } }
2497   }
2498 }

```

## 2.15 The glyphs of the 10 arabic numbers

```

2499 \cs_new_protected:Npn \@@_draw_number:n #1
2500 { \tl_map_inline:nn { #1 } { \use:c { c_@@_glyph_ ##1 _tl } } }
2501 \cs_generate_variant:Nn \@@_draw_number:n { e }

2502 \cs_set_eq:NN \tlconstcn \tl_const:cn

2503 \ExplSyntaxOff
2504 \tlconstcn{c_@@_glyph_0_tl}
2505 {%
2506   \hbox to 5pt
2507   {%
2508     \hfill
2509     \pdfextension literal
2510     {
2511       4.24 3.05 m
2512       4.24 4.91 3.22 6.22 2.12 6.22 c
2513       1.00 6.22 0 4.88 0 3.06 c
2514       0 1.20 1.02 -0.11 2.12 -0.11 c
2515       3.24 -0.11 4.24 1.23 4.24 3.05 c
2516       3.55 3.16 m
2517       3.55 1.68 2.90 0.50 2.12 0.50 c
2518       1.34 0.50 0.69 1.68 0.69 3.16 c
2519       0.69 4.62 1.38 5.61 2.12 5.61 c
2520       2.85 5.61 3.55 4.63 3.55 3.16 c
2521       h f
2522     }%
2523     \hfill
2524   }%
2525 }

2526 \tlconstcn{c_@@_glyph_1_tl}
2527 {%
2528   \hbox to 5pt
2529   {%
2530     \hfill \kern 1 pt
2531     \pdfextension literal
2532     {
2533       3.37 0.3 m
2534       3.37 0.61 3.13 0.61 2.97 0.61 c
2535       2.06 0.61 l
2536       2.06 5.81 l
2537       2.06 5.97 2.06 6.22 1.76 6.22 c
2538       1.57 6.22 1.51 6.1 1.46 5.98 c
2539       1.08 5.13 0.56 5.02 0.37 5.0 c
2540       0.21 4.99 0.0 4.97 0.0 4.69 c
2541       0.0 4.44 0.18 4.39 0.33 4.39 c
2542       0.52 4.39 0.93 4.45 1.37 4.83 c
2543       1.37 0.61 l
2544       0.46 0.61 l
2545       0.3 0.61 0.06 0.61 0.06 0.3 c
2546       0.06 0.0 0.31 0.0 0.46 0.0 c
2547       2.97 0.0 l
2548       3.12 0.0 3.37 0.0 3.37 0.3 c
2549       h f

```

```

2550         }%
2551     \hfill
2552 }%
2553 }

2554 \tlconstcn{c_@@_glyph_2_tl}
2555 {%
2556     \hbox to 5pt
2557     {%
2558         \hfill
2559         \pdfextension literal
2560         {
2561             4.2 0.41 m
2562             4.2 0.67 l
2563             4.2 0.86 4.2 1.08 3.86 1.08 c
2564             3.51 1.08 3.51 0.89 3.51 0.61 c
2565             1.13 0.61 l
2566             1.72 1.12 2.68 1.87 3.11 2.27 c
2567             3.74 2.83 4.2 3.47 4.2 4.27 c
2568             4.2 5.47 3.19 6.22 1.97 6.22 c
2569             0.79 6.22 0.0 5.4 0.0 4.55 c
2570             0.0 4.18 0.28 4.07 0.45 4.07 c
2571             0.66 4.07 0.89 4.24 0.89 4.52 c
2572             0.89 4.64 0.84 4.77 0.75 4.84 c
2573             0.9 5.3 1.37 5.61 1.92 5.61 c
2574             2.74 5.61 3.51 5.15 3.51 4.27 c
2575             3.51 3.57 3.02 2.99 2.36 2.44 c
2576             0.15 0.58 l
2577             0.06 0.5 0.0 0.45 0.0 0.31 c
2578             0.0 0.0 0.25 0.0 0.41 0.0 c
2579             3.8 0.0 l
2580             4.13 0.0 4.2 0.09 4.2 0.41 c
2581         h f
2582     }%
2583     \hfill
2584 }%
2585 }

2586 \tlconstcn{c_@@_glyph_3_tl}
2587 {%
2588     \hbox to 5pt
2589     {%
2590         \hfill
2591         \pdfextension literal
2592         {
2593             4.36 1.74 m
2594             4.36 2.45 3.89 3.04 3.23 3.34 c
2595             3.79 3.7 4.07 4.27 4.07 4.81 c
2596             4.07 5.55 3.34 6.22 2.19 6.22 c
2597             0.99 6.22 0.29 5.74 0.29 5.05 c
2598             0.29 4.72 0.54 4.58 0.74 4.58 c
2599             0.95 4.58 1.18 4.75 1.18 5.03 c
2600             1.18 5.17 1.12 5.27 1.09 5.3 c
2601             1.4 5.61 2.11 5.61 2.2 5.61 c
2602             2.88 5.61 3.38 5.25 3.38 4.8 c
2603             3.38 4.5 3.23 4.15 2.96 3.93 c
2604             2.64 3.67 2.39 3.65 2.03 3.63 c
2605             1.46 3.59 1.31 3.59 1.31 3.3 c
2606             1.31 2.99 1.55 2.99 1.71 2.99 c
2607             2.17 2.99 l
2608             3.16 2.99 3.67 2.32 3.67 1.74 c
2609             3.67 1.13 3.11 0.5 2.2 0.5 c
2610             1.8 0.5 1.03 0.61 0.77 1.08 c
2611             0.82 1.13 0.89 1.19 0.89 1.39 c
2612             0.89 1.63 0.7 1.83 0.45 1.83 c

```



```

2613         0.22 1.83 0.0 1.68 0.0 1.36 c
2614         0.0 0.47 0.97 -0.11 2.2 -0.11 c
2615         3.51 -0.11 4.36 0.81 4.36 1.74 c
2616         h f
2617     }%
2618     \hfill
2619 }%
2620 }

2621 \tlconstcn{c_@@_glyph_4_tl}
2622 {%
2623     \hbox to 5pt
2624     {%
2625         \hfill
2626         \pdfextension literal
2627         {
2628             4.66 1.99 m
2629             4.66 2.3 4.42 2.3 4.26 2.3 c
2630             3.48 2.3 l
2631             3.48 5.82 l
2632             3.48 6.15 3.41 6.23 3.07 6.23 c
2633             2.79 6.23 l
2634             2.55 6.23 2.5 6.22 2.37 6.02 c
2635             0.09 2.44 l
2636             0.0 2.31 0.0 2.29 0.0 2.09 c
2637             0.0 1.76 0.09 1.69 0.4 1.69 c
2638             2.92 1.69 l
2639             2.92 0.61 l
2640             2.3 0.61 l
2641             2.14 0.61 1.9 0.61 1.9 0.3 c
2642             1.9 0.0 2.15 0.0 2.3 0.0 c
2643             4.1 0.0 l
2644             4.25 0.0 4.5 0.0 4.5 0.3 c
2645             4.5 0.61 4.26 0.61 4.1 0.61 c
2646             3.48 0.61 l
2647             3.48 1.69 l
2648             4.26 1.69 l
2649             4.41 1.69 4.66 1.69 4.66 1.99 c
2650             2.92 2.3 m
2651             0.7 2.3 l
2652             2.92 5.79 l
2653             h f
2654         }%
2655         \hfill
2656     }%
2657 }

2658 \tlconstcn{c_@@_glyph_5_tl}
2659 {%
2660     \hbox to 5pt
2661     {%
2662         \hfill
2663         \pdfextension literal
2664         {
2665             4.2 1.9 m
2666             4.2 2.91 3.43 3.89 2.25 3.89 c
2667             1.9 3.89 1.46 3.82 1.05 3.6 c
2668             1.05 5.5 l
2669             3.45 5.5 l
2670             3.6 5.5 3.85 5.5 3.85 5.8 c
2671             3.85 6.11 3.61 6.11 3.45 6.11 c
2672             0.76 6.11 l
2673             0.43 6.11 0.36 6.02 0.36 5.7 c
2674             0.36 3.04 l
2675             0.36 2.86 0.36 2.63 0.68 2.63 c

```

```

2676         0.86 2.63 0.9 2.68 0.98 2.78 c
2677         1.25 3.1 1.66 3.28 2.24 3.28 c
2678         3.06 3.28 3.51 2.55 3.51 1.9 c
2679         3.51 1.1 2.8 0.5 1.96 0.5 c
2680         1.68 0.5 1.04 0.58 0.76 1.13 c
2681         0.81 1.18 0.89 1.25 0.89 1.45 c
2682         0.89 1.74 0.66 1.9 0.45 1.9 c
2683         0.3 1.9 0.0 1.81 0.0 1.42 c
2684         0.0 0.59 0.84 -0.11 1.96 -0.11 c
2685         3.21 -0.11 4.2 0.79 4.2 1.9 c
2686         h f
2687     }%
2688 \hfill
2689 }%
2690 }

2691 \tlconstcn{c_@@_glyph_6_tl}
2692 {%
2693 \hbox to 5pt
2694 {%
2695 \hfill
2696 \pdfextension literal
2697 {
2698     4.18 1.93 m
2699     4.18 3.07 3.3 3.96 2.2 3.96 c
2700     1.68 3.96 1.16 3.79 0.71 3.38 c
2701     0.85 4.79 1.79 5.61 2.67 5.61 c
2702     3.01 5.61 3.17 5.5 3.22 5.45 c
2703     3.17 5.4 3.12 5.34 3.12 5.16 c
2704     3.12 4.92 3.3 4.72 3.56 4.72 c
2705     3.81 4.72 4.01 4.89 4.01 5.19 c
2706     4.01 5.68 3.65 6.22 2.68 6.22 c
2707     1.34 6.22 0.0 5.01 0.0 2.99 c
2708     0.0 0.62 1.12 -0.11 2.11 -0.11 c
2709     3.2 -0.11 4.18 0.73 4.18 1.93 c
2710     3.49 1.93 m
2711     3.49 1.07 2.83 0.5 2.11 0.5 c
2712     1.46 0.5 1.07 0.99 0.87 1.6 c
2713     0.77 1.84 0.79 2.08 0.79 2.22 c
2714     0.79 2.84 1.39 3.34 2.15 3.34 c
2715     2.96 3.34 3.49 2.67 3.49 1.93 c
2716     h f
2717 }%
2718 \hfill
2719 }%
2720 }

2721 \tlconstcn{c_@@_glyph_7_tl}
2722 {%
2723 \hbox to 5pt
2724 {%
2725 \hfill
2726 \pdfextension literal
2727 {
2728     4.36 5.8 m
2729     4.36 6.11 4.12 6.11 3.96 6.11 c
2730     0.66 6.11 l
2731     0.6 6.27 0.42 6.27 0.34 6.27 c
2732     0.0 6.27 0.0 6.05 0.0 5.86 c
2733     0.0 5.44 l
2734     0.0 5.25 0.0 5.03 0.34 5.03 c
2735     0.69 5.03 0.69 5.22 0.69 5.5 c
2736     3.33 5.5 l
2737     1.6 3.66 1.26 1.46 1.26 0.36 c
2738     1.26 0.22 1.26 -0.11 1.61 -0.11 c

```

```

2739         1.78 -0.11 1.95 0.0 1.95 0.29 c
2740         2.0 3.13 3.53 4.87 4.14 5.46 c
2741         4.34 5.64 4.36 5.66 4.36 5.8 c
2742         h f
2743     }%
2744     \hfill
2745 }%
2746 }

2747 \tlconstcn{c_@@_glyph_8_tl}
2748 {%
2749     \hbox to 5pt
2750     {%
2751         \hfill
2752         \pdfextension literal
2753         {
2754             4.36 1.74 m
2755             4.36 2.49 3.71 3.08 2.97 3.29 c
2756             3.77 3.56 4.22 4.05 4.22 4.62 c
2757             4.22 5.44 3.37 6.22 2.18 6.22 c
2758             0.98 6.22 0.14 5.43 0.14 4.62 c
2759             0.14 4.05 0.6 3.55 1.39 3.29 c
2760             0.64 3.08 0.0 2.49 0.0 1.74 c
2761             0.0 0.77 0.93 -0.11 2.18 -0.11 c
2762             3.44 -0.11 4.36 0.77 4.36 1.74 c
2763             3.53 4.61 m
2764             3.53 4.04 2.91 3.6 2.18 3.6 c
2765             1.45 3.6 0.83 4.05 0.83 4.61 c
2766             0.83 5.13 1.39 5.61 2.18 5.61 c
2767             2.96 5.61 3.53 5.13 3.53 4.61 c
2768             3.67 1.75 m
2769             3.67 1.06 3.01 0.5 2.18 0.5 c
2770             1.35 0.5 0.69 1.06 0.69 1.75 c
2771             0.69 2.36 1.27 2.99 2.18 2.99 c
2772             3.1 2.99 3.67 2.36 3.67 1.75 c
2773             h f
2774         }%
2775         \hfill
2776     }%
2777 }

2778 \tlconstcn{c_@@_glyph_9_tl}
2779 {%
2780     \hbox to 5pt
2781     {%
2782         \hfill
2783         \pdfextension literal
2784         {
2785             4.18 3.12 m
2786             4.18 5.53 3.07 6.22 2.12 6.22 c
2787             1.01 6.22 0.0 5.39 0.0 4.18 c
2788             0.0 3.04 0.88 2.15 1.98 2.15 c
2789             2.5 2.15 3.02 2.32 3.47 2.73 c
2790             3.37 1.49 2.59 0.5 1.63 0.5 c
2791             1.54 0.5 1.18 0.51 0.97 0.67 c
2792             1.01 0.72 1.06 0.78 1.06 0.95 c
2793             1.06 1.19 0.88 1.39 0.62 1.39 c
2794             0.37 1.39 0.17 1.22 0.17 0.92 c
2795             0.17 0.6 0.35 -0.11 1.63 -0.11 c
2796             2.95 -0.11 4.18 1.14 4.18 3.12 c
2797             3.4 3.89 m
2798             3.4 3.34 2.86 2.77 2.03 2.77 c
2799             1.22 2.77 0.69 3.44 0.69 4.18 c
2800             0.69 5.05 1.4 5.61 2.12 5.61 c
2801             2.55 5.61 2.83 5.38 2.99 5.18 c

```

```

2802         3.31 4.79 3.4 4.6 3.4 3.89 c
2803         h f
2804     }%
2805     \hfill
2806 }%
2807 }
2808 \ExplSyntaxOn
2809 % \end{macrocode}
2810 %
2811 % \bigskip
2812 % \subsection{We load piton.lua}
2813 %
2814 %
2815 % \bigskip
2816 % \begin{macrocode}
2817 \cs_new_protected:Npn \@@_test_version:n #1
2818 {
2819     \str_if_eq:onF \PitonFileVersion { #1 }
2820     { \@@_error:n { bad-version-of-piton.lua } }
2821 }

2822 \hook_gput_code:nnn { begindocument } { . }
2823 {
2824     \lua_load_module:n { piton }
2825     \lua_now:n
2826     {
2827         tex.sprint ( luatexbase.catcodetables.expl ,
2828                     [[\@@_test_version:n {}] .. piton_version .. "]" )
2829     }
2830 }
</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`.

```

2831 ⟨*LUA⟩
2832 piton.comment_latex = piton.comment_latex or ">"
2833 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`.

```

2834 piton.write_files = { }
2835 piton.join_files = { }

2836 local sprintL3
2837 function sprintL3 ( s )
2838     tex.sprint ( luatexbase.catcodetables.expl , s )
2839 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.

```
2840 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2841 local Cg, Cmt, Cb = lpeg.Cg, lpeg.Cmt, lpeg.Cb
2842 local B, R = lpeg.B, lpeg.R
```

The following line is mandatory.

```
2843 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”).

```
2844 local Q
2845 function Q ( pattern )
2846   return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2847 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.

```
2848 local L
2849 function L ( pattern ) return
2850   Ct ( C ( pattern ) )
2851 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.

```
2852 local Lc
2853 function Lc ( string ) return
2854   Cc ( { luatexbase.catcodetables.expl, string } )
2855 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)

```
2856 local K
2857 function K ( style, pattern ) return
2858   Lc ( [ [ {\PitonStyle{ }} .. style .. "}{"} ] ]
2859   * Q ( pattern )
2860   * Lc "}" ] ]
2861 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.

```
2862 local WithStyle
2863 function WithStyle ( style , pattern ) return
2864     Ct ( Cc "Open" * Cc ( [[{\PitonStyle{}}] .. style .. "}{"} ) * Cc "}" )
2865     * pattern
2866     * Ct ( Cc "Close" )
2867 end
```

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

```
2868 Escape = P ( false )
2869 EscapeClean = P ( false )
2870 if piton.begin_escape then
2871     Escape =
2872         P ( piton.begin_escape )
2873         * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
2874         * 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).

```
2875     EscapeClean =
2876         P ( piton.begin_escape )
2877         * ( 1 - P ( piton.end_escape ) ) ^ 1
2878         * P ( piton.end_escape )
2879 end

2880 EscapeMath = P ( false )
2881 if piton.begin_escape_math then
2882     EscapeMath =
2883         P ( piton.begin_escape_math )
2884         * Lc "$"
2885         * L ( ( 1 - P(piton.end_escape_math) ) ^ 1 )
2886         * Lc "$"
2887         * P ( piton.end_escape_math )
2888 end
```

## The basic syntactic LPEG

```
2889 local alpha , digit = lpeg.alpha , lpeg.digit
2890 local space = P " "
```

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

```
2891 local letter = alpha + "_" + "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "î"
2892               + "ô" + "û" + "ü" + "Ë" + "Ä" + "Å" + "Ç" + "É" + "Ê" + "Ë" + "Ï"
2893               + "Î" + "Ï" + "Œ" + "Ů" + "Ů"
2894
2895 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).

```
2896 local identifier = letter * alphanum ^ 0
```

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

```
2897 local Identifier = K ( 'Identifier.Internal' , identifier )
```

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

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

```

2898 local allow_underscores_except_first
2899 function allow_underscores_except_first ( p )
2900     return p * (P "_" + p)^0
2901 end
2902 local allow_underscores
2903 function allow_underscores ( p )
2904     return (P "_" + p)^0
2905 end
2906 local digits_to_number
2907 function digits_to_number(prefix, digits)
2908     -- The edge cases of what is allowed in number literals is modelled after
2909     -- OCaml numbers, which seems to be the most permissive language
2910     -- in this regard (among C, OCaml, Python & SQL).
2911     return prefix
2912         * allow_underscores_except_first(digits^1)
2913         * (P "." * #(1 - P ".") * allow_underscores(digits))^~1
2914         * (S "eE" * S "+-")^~1 * allow_underscores_except_first(digits^1))^~1
2915 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).

```

2916 local Number =
2917     K ( 'Number.Internal' ,
2918         digits_to_number (P "0x" + P "OX", R "af" + R "AF" + digit)
2919         + digits_to_number (P "0o" + P "OO", R "07")
2920         + digits_to_number (P "0b" + P "OB", R "01")
2921         + digits_to_number ( "" , digit )
2922     )

```

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?).

```

2923 local lpeg_central = 1 - S " '\r[({}])'" - digit

```

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

```

2924 if piton.begin_escape then
2925     lpeg_central = lpeg_central - piton.begin_escape
2926 end
2927 if piton.begin_escape_math then
2928     lpeg_central = lpeg_central - piton.begin_escape_math
2929 end
2930 local Word = Q ( lpeg_central ^ 1 )

2931 local Space = Q " " ^ 1
2932 local SkipSpace = Q " " ^ 0
2933
2934 local Punct = Q ( S ".,:;!)"
2935
2936 local Tab = "\t" * Lc [[ \@_tab: ]]

```

---

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

```
2937 local LeadingSpace = Lc [[ \@@_leading_space: ]] * P " "
```

```
2938 local Delim = Q ( S "[({})]" )
```

The following LPEG catches a space (U+0020) and replaces it by `\l_@@_space_in_string_tl`. It will be used in the strings. Usually, `\l_@@_space_in_string_tl` 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_tl` will contain `␣` (U+2423) in order to visualize the spaces.

```
2939 local SpaceInString = space * Lc [[ \l_@@_space_in_string_tl ]]
```

### 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).

```
2940 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
2941 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
2942 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
2943 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.

```
2944 local detectedCommands = P ( false )
2945 for _ , x in ipairs ( detected_commands ) do
2946   detectedCommands = detectedCommands + P ( "\\\" .. x )
2947 end
```

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

```
2948 local rawDetectedCommands = P ( false )
2949 for _ , x in ipairs ( raw_detected_commands ) do
2950   rawDetectedCommands = rawDetectedCommands + P ( "\\\" .. x )
2951 end

2952 local beamerCommands = P ( false )
2953 for _ , x in ipairs ( beamer_commands ) do
2954   beamerCommands = beamerCommands + P ( "\\\" .. x )
2955 end

2956 local beamerEnvironments = P ( false )
2957 for _ , x in ipairs ( beamer_environments ) do
2958   beamerEnvironments = beamerEnvironments + P ( x )
2959 end
```



## Several tools for the construction of the main LPEG

```
2960 local LPEG0 = { }
2961 local LPEG1 = { }
2962 local LPEG2 = { }
2963 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*.

```
2964 local Compute_braces
2965 function Compute_braces ( lpeg_string ) return
2966   P { "E" ,
2967     E =
2968       (
2969         "{" * V "E" * "}"
2970       +
2971         lpeg_string
2972       +
2973         ( 1 - S "{" )
2974       ) ^ 0
2975   }
2976 end
```

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

```
2977 local Compute_DetectedCommands
2978 function Compute_DetectedCommands ( lang , braces ) return
2979   Ct (
2980     Cc "Open"
2981     * C ( detectedCommands * space ^ 0 * P "{" )
2982     * Cc "}"
2983   )
2984   * ( braces
2985     / ( function ( s )
2986         if s ~= '' then return
2987           LPEG1[lang] : match ( s )
2988         end
2989       end )
2990   )
2991   * P "}"
2992   * Ct ( Cc "Close" )
2993 end
2994 local Compute_RawDetectedCommands
2995 function Compute_RawDetectedCommands ( lang , braces ) return
2996   Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
2997 end
2998 local Compute_LPEG_cleaner
2999 function Compute_LPEG_cleaner ( lang , braces ) return
3000   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
3001     * ( braces
3002       / ( function ( s )
3003         if s ~= '' then return
3004           LPEG_cleaner[lang] : match ( s )
3005         end
3006       end )
3007     )
3008     * "}"
3009     + EscapeClean
```

```

3010         + C ( P ( 1 ) )
3011     ) ^ 0 ) / table.concat
3012 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`.

```

3013 local ParseAgain
3014 function ParseAgain ( code )
3015     if code ~= '' then return

```

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

```

3016     LPEG1[piton.language] : match ( code )
3017 end
3018 end

```

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

```

3019 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.

```

3020 local Compute_Beamer
3021 function Compute_Beamer ( lang , braces )

```

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

```

3022     local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
3023     lpeg = lpeg +
3024         Ct ( Cc "Open"
3025             * C ( beamerCommands
3026                 * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3027                 * P "{"
3028             )
3029             * Cc "}"
3030         )
3031     * ( braces /
3032         ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3033     * "]"
3034     * Ct ( Cc "Close" )

```

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

```

3035     lpeg = lpeg +
3036         L ( P [[\alt]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
3037         * ( braces /
3038             ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3039         * L ( P "}" )
3040         * ( braces /
3041             ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3042         * L ( P "}" )

```

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

```

3043 lpeg = lpeg +
3044   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
3045   * ( braces
3046     / ( function ( s )
3047       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3048   * L ( P "}{" )
3049   * ( braces
3050     / ( function ( s )
3051       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3052   * L ( P "}{" )
3053   * ( braces
3054     / ( function ( s )
3055       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3056   * L ( P "}" )

```

Now, the environments of Beamer.

```

3057 for _ , x in ipairs ( beamer_environments ) do
3058   lpeg = lpeg +
3059     Ct ( Cc "Open"
3060         * C (
3061           P ( [[\begin{]] .. x .. "]" )
3062             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3063           )
3064           * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
3065           * Cc ( [[\end{]] .. x .. "]" )
3066         )
3067     * (

```

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).

```

3068       (
3069         P { "E" ,
3070           E = (
3071             P ( [[\begin{]] .. x .. "]" )
3072             * V "E"
3073             * P ( [[\end{]] .. x .. "]" )
3074             +
3075             (
3076               1
3077               - P ( [[\begin{]] .. x .. "]" )
3078               - P ( [[\end{]] .. x .. "]" )
3079             )
3080             ) ^ 0
3081           }
3082       )
3083       / ( function ( s )
3084         if s ~= '' then return
3085           LPEG1[lang] : match ( s )
3086         end
3087       end )
3088     )
3089   * P ( [[\end{]] .. x .. "]" )
3090   * Ct ( Cc "Close" )
3091 end

```

Now, you can return the value we have computed.

```

3092   return lpeg
3093 end

```

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

```

3094 local CommentMath =
3095   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`.

```
3096 local Prompt =
3097   K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
3098   * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]
```

The following LPEG EOL is for the end of lines.

```
3099 local EOL =
3100   P "\r"
3101   *
3102   (
3103     space ^ 0 * -1
3104     +
3105     Cc "EOL"
3106   )
3107   * ( LeadingSpace ^ 0 * # ( 1 - S " \r" ) ) ^ -1
```

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

```
3108 local CommentLaTeX =
3109   P ( piton.comment_latex )
3110   * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}]]
3111   * L ( ( 1 - P "\r" ) ^ 0 )
3112   * Lc "}"
3113   * ( 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).

```
3114 --python Python
3115 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.

```
3116 local Operator =
3117   K ( 'Operator' ,
3118     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "/" + "*"
3119     + S "--+/*%=<>&.@|" )
3120
3121 local OperatorWord =
3122   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`.

```
3123 local For = K ( 'Keyword' , P "for" )
3124   * Space
3125   * Identifier
3126   * Space
3127   * K ( 'Keyword' , P "in" )
3128
3129 local Keyword =
3130   K ( 'Keyword' ,
3131     P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
3132     "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
3133     "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
3134     "try" + "while" + "with" + "yield" + "yield from" )
3135   + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
3136
3137 local Builtin =
3138   K ( 'Name.Builtin' ,
```

```

3139 P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
3140 "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
3141 "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
3142 "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
3143 "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
3144 "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
3145 + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
3146 "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
3147 "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
3148 "vars" + "zip" )
3149
3150 local Exception =
3151   K ( 'Exception' ,
3152     P "ArithmeticError" + "AssertionError" + "AttributeError" +
3153     "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
3154     "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
3155     "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
3156     "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
3157     "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
3158     "NotImplementedError" + "OSError" + "OverflowError" +
3159     "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
3160     "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
3161     "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
3162 + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
3163 "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
3164 "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
3165 "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
3166 "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
3167 "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
3168 "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
3169 "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
3170 "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
3171 "RecursionError" )
3172
3173 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.

```

3174 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:`

```

3175 local DefClass =
3176   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`

```

3177 local ImportAs =
3178   K ( 'Keyword' , "import" )
3179   * Space
3180   * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
3181   * (

```

```

3182      ( Space * K ( 'Keyword' , "as" ) * Space
3183        * K ( 'Name.Namespace' , identifier ) )
3184      +
3185      ( SkipSpace * Q "," * SkipSpace
3186        * K ( 'Name.Namespace' , identifier ) ) ^ 0
3187    )

```

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`

```

3188  local FromImport =
3189    K ( 'Keyword' , "from" )
3190      * Space * K ( 'Name.Namespace' , identifier )
3191      * 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 instruction<sup>4</sup> in that interpolation:

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

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

```

3192  local PercentInterpol =
3193    K ( 'String.Interpol' ,
3194      P "%"
3195      * ( "(" * alphanum ^ 1 * ")" ) ^ -1
3196      * ( S "-#0 +" ) ^ 0
3197      * ( digit ^ 1 + "*" ) ^ -1
3198      * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
3199      * ( S "HLL" ) ^ -1
3200      * S "sdfFeExXorgiGauc%"
3201    )

```

We can now define the LPEG for the four kinds of strings. It's not possible to use our function `K` because of the interpolations which must be formatted with another `piton` style that the rest of the string.<sup>5</sup>

```

3202  local SingleShortString =
3203    WithStyle ( 'String.Short.Internal' ,

```

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

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

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

```

3204     Q ( P "f'" + "F'" )
3205     * (
3206         K ( 'String.Interpol' , "{" )
3207         * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0 )
3208         * Q ( P ":" * ( 1 - S "':" ) ^ 0 ) ^ -1
3209         * K ( 'String.Interpol' , "}" )
3210         +
3211         SpaceInString
3212         +
3213         Q ( ( P "\\'" + "\\\"" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
3214     ) ^ 0
3215     * Q ""
3216 +

```

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

```

3217     Q ( P "'" + "r'" + "R'" )
3218     * ( Q ( ( P "\\'" + "\\\"" + 1 - S " \'r%" ) ^ 1 )
3219         + SpaceInString
3220         + PercentInterpol
3221         + Q "%"
3222     ) ^ 0
3223     * Q "" )
3224 local DoubleShortString =
3225     WithStyle ( 'String.Short.Internal' ,
3226         Q ( P "f\"" + "F\"" )
3227         * (
3228             K ( 'String.Interpol' , "{" )
3229             * K ( 'Interpol.Inside' , ( 1 - S "}\" )" ) ^ 0 )
3230             * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S ":\") ^ 0 ) ) ^ -1
3231             * K ( 'String.Interpol' , "}" )
3232             +
3233             SpaceInString
3234             +
3235             Q ( ( P "\\\"" + "\\\"" + "{{" + "}" + 1 - S " {}\" )" ^ 1 )
3236         ) ^ 0
3237         * Q "\"
3238     +
3239     Q ( P "\" + "r\"" + "R\"" )
3240     * ( Q ( ( P "\\\"" + "\\\"" + 1 - S " \"r%" ) ^ 1 )
3241         + SpaceInString
3242         + PercentInterpol
3243         + Q "%"
3244     ) ^ 0
3245     * Q "\" )
3246
3247 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.

```

3248 local braces =
3249     Compute_braces
3250     (
3251         ( P "\" + "r\"" + "R\"" + "f\"" + "F\"" )
3252         * ( P '\\\"' + 1 - S "\\\" ) ^ 0 * "\"
3253     +
3254         ( P '\'' + 'r\'' + 'R\'' + 'f\'' + 'F\'' )
3255         * ( P '\\\'' + 1 - S '\\\'' ) ^ 0 * '\''
3256     )
3257
3258 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

## Detected commands

```
3259 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
3260       + Compute_RawDetectedCommands ( 'python' , braces )
```

## LPEG\_cleaner

```
3261 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )
```

## The long strings

```
3262 local SingleLongString =
3263   WithStyle ( 'String.Long.Internal' ,
3264     ( Q ( S "fF" * P "'''' " )
3265       * (
3266         K ( 'String.Interpol' , "{" )
3267         * K ( 'Interpol.Inside' , ( 1 - S "j:\r" - "'''' " ) ^ 0 )
3268         * Q ( P ":" * ( 1 - S "j:\r" - "'''' " ) ^ 0 ) ^ -1
3269         * K ( 'String.Interpol' , "}" )
3270       +
3271         Q ( ( 1 - P "'''' " - S "{}'\r" ) ^ 1 )
3272       +
3273         EOL
3274     ) ^ 0
3275   +
3276   Q ( ( S "rR" ) ^ -1 * "'''' " )
3277   * (
3278     Q ( ( 1 - P "'''' " - S "\r%" ) ^ 1 )
3279     +
3280     PercentInterpol
3281     +
3282     P "%"
3283     +
3284     EOL
3285   ) ^ 0
3286 )
3287 * Q "'''' " )

3288 local DoubleLongString =
3289   WithStyle ( 'String.Long.Internal' ,
3290     (
3291       Q ( S "fF" * "\"\"\"\" " )
3292       * (
3293         K ( 'String.Interpol' , "{" )
3294         * K ( 'Interpol.Inside' , ( 1 - S "j:\r" - "\"\"\"\" " ) ^ 0 )
3295         * Q ( ":" * ( 1 - S "j:\r" - "\"\"\"\" " ) ^ 0 ) ^ -1
3296         * K ( 'String.Interpol' , "}" )
3297       +
3298         Q ( ( 1 - S "{}\"\"\"\" - "\"\"\"\" " ) ^ 1 )
3299       +
3300         EOL
3301     ) ^ 0
3302   +
3303   Q ( S "rR" ^ -1 * "\"\"\"\" " )
3304   * (
3305     Q ( ( 1 - P "\"\"\"\" - S "%\r" ) ^ 1 )
3306     +
3307     PercentInterpol
3308     +
3309     P "%"
3310     +
3311     EOL
3312   ) ^ 0
```



```

3313     )
3314     * Q "\"\"\""
3315 )
3316 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`).

```

3317 local StringDoc =
3318     K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\"\"" )
3319     * ( K ( 'String.Doc.Internal' , ( 1 - P "\"\"\"" - "r" ) ^ 0 ) * EOL
3320         * Tab ^ 0
3321     ) ^ 0
3322     * 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.

```

3323 local Comment =
3324     WithStyle
3325     ( 'Comment.Internal' ,
3326     Q "#" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3327     )
3328     * ( 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).

```

3329 local expression =
3330     P { "E" ,
3331         E = ( "'" * ( P "\\'" + 1 - S "'\r" ) ^ 0 * "'"
3332             + "\"" * ( P "\\\"" + 1 - S "\"\r" ) ^ 0 * "\""
3333             + "{" * V "F" * "}"
3334             + "(" * V "F" * ")"
3335             + "[" * V "F" * "]"
3336             + ( 1 - S "{}()[]\r," ) ^ 0 ,
3337         F = ( "{" * V "F" * "}"
3338             + "(" * V "F" * ")"
3339             + "[" * V "F" * "]"
3340             + ( 1 - S "{}()[]\r\\"" ) ^ 0
3341     }

```

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`.

```

3342 local Params =
3343     P { "E" ,
3344         E = ( V "F" * ( Q "," * V "F" ) ^ 0 ) ^ -1 ,
3345         F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
3346             * ( Q ":" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3347             * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
3348     }

```

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 `python.sty`) after the definition of several other LPEG such as `Comment`, `CommentLaTeX`, `Params`, `StringDoc`...

```

3349 local DefFunction =
3350   K ( 'Keyword' , "def" )
3351   * Space
3352   * K ( 'Name.Function.Internal' , identifier )
3353   * SkipSpace
3354   * Q "(" * Params * Q ")"
3355   * SkipSpace
3356   * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3357   * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
3358   * Q ":"
3359   * ( SkipSpace
3360     * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
3361     * Tab ^ 0
3362     * SkipSpace
3363     * StringDoc ^ 0 -- there may be additional docstrings
3364   ) ^ -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

```

3365 local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

## The main LPEG for the language Python

```

3366 local EndKeyword
3367   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3368     EscapeMath + -1

```

First, the main loop :

```

3369 local Main =
3370   space ^ 0 * EOL
3371   + Space
3372   + Tab
3373   + Escape + EscapeMath
3374   + Beamer
3375   + CommentLaTeX
3376   + DetectedCommands
3377   + Prompt
3378   + LongString
3379   + Comment
3380   + ExceptionInConsole
3381   + Delim
3382   + Operator
3383   + OperatorWord * EndKeyword
3384   + ShortString
3385   + Punct
3386   + FromImport
3387   + RaiseException
3388   + DefFunction
3389   + DefClass
3390   + For
3391   + Keyword * EndKeyword
3392   + Decorator
3393   + Builtin * EndKeyword

```

```

3394      + Identifier
3395      + Number
3396      + Word

```

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

```

3397  LPEG1.python = Main ^ 0

```

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

```

3398  LPEG2.python =
3399      Ct (
3400          ( space ^ 0 * "\r" ) ^ -1
3401          * Lc [[ \@@_begin_line: ]]
3402          * LeadingSpace ^ 0
3403          * ( space ^ 1 * -1 + Main ) ^ 0
3404          * -1
3405          * Lc [[ \@@_end_line: ]]
3406      )

```

End of the Lua scope for the language Python.

```

3407  end

```

### 3.5 The language OCaml

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

```

3408  --ocaml Ocaml OCaml
3409  do

3410      local SkipSpace = ( Q " " + EOL ) ^ 0
3411      local Space = ( Q " " + EOL ) ^ 1

3412      local braces = Compute_braces ( '\"' * ( 1 - S "\"" ) ^ 0 * '\"' )

3413      if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
3414      DetectedCommands =
3415          Compute_DetectedCommands ( 'ocaml' , braces )
3416          + Compute_RawDetectedCommands ( 'ocaml' , braces )
3417      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`.

```

3418  function Q ( pattern, strict )
3419      if strict ~= nil then
3420          return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3421      else
3422          return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3423              + Beamer + DetectedCommands + EscapeMath + Escape
3424      end
3425  end

```

---

<sup>6</sup>Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

```

3426 local K
3427 function K ( style , pattern, strict ) return
3428   Lc ( [[ {\PitonStyle{ }} .. style .. "}{" )
3429   * Q ( pattern, strict )
3430   * Lc "}" }"
3431 end

3432 local WithStyle
3433 function WithStyle ( style , pattern ) return
3434   Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{ }} .. style .. "}{" ) * Cc "}" }" )
3435   * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
3436   * Ct ( Cc "Close" )
3437 end

```

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

```

3438 local balanced_parens =
3439   P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "()") ) ^ 0 }

```

### The strings of OCaml

```

3440 local ocaml_string =
3441   P "\"\"
3442   * (
3443     P " "
3444     +
3445     P ( ( P '\\\"' + 1 - S "\\r" ) ^ 1 )
3446     +
3447     EOL -- ?
3448   ) ^ 0
3449   * P "\"\"

3450 local String =
3451   WithStyle
3452   ( 'String.Long.Internal' ,
3453     Q "\"\"
3454     * (
3455       SpaceInString
3456       +
3457       Q ( ( P '\\\"' + 1 - S "\\r" ) ^ 1 )
3458       +
3459       EOL
3460     ) ^ 0
3461     * Q "\"\"
3462   )

```

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

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

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

```

3463 local ext = ( R "az" + "_" ) ^ 0
3464 local open = "{" * Cg ( ext , 'init' ) * "/"
3465 local close = "/" * C ( ext ) * "}"
3466 local closeeq =
3467   Cmt ( close * Cb ( 'init' ) ,
3468     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3469   local QuotedStringBis =
3470     WithStyle ( 'String.Long.Internal' ,
3471       (
3472         Space
3473         +
3474         Q ( ( 1 - S " \r" ) ^ 1 )
3475         +
3476         EOL
3477       ) ^ 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.

```

3478   local QuotedString =
3479     C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3480     ( 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).

```

3481   local comment =
3482     P {
3483       "A" ,
3484       A = Q "(*"
3485         * ( V "A"
3486           + Q ( ( 1 - S "\r$\\"" - "(*" - "*)" ) ^ 1 ) -- $
3487           + Q ( ocaml_string )
3488           + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3489           + EOL
3490         ) ^ 0
3491       * Q "*)"
3492     }
3493   local Comment = WithStyle ( 'Comment.Internal' , comment )

```

## Some standard LPEG

```

3494   local Delim = K ( 'Delim' , P "[|" + "|]" + S "[()]" )
3495   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.

```

3496   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.

```

3497   local Constructor =
3498     P "::"

```

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

```

3499   * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3500   +
3501   P "[]"
3502   * Lc [[{\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]]]
3503   K ( 'Name.Constructor' ,
3504     Q "`" ^ -1 * cap_identifier
3505     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )
3506   local ModuleType = K ( 'Name.Type' , cap_identifier )

```

```

3507 local OperatorWord =
3508   K ( 'Operator.Word' ,
3509       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )

```

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

```

3510 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3511   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3512   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3513   "struct" + "type" + "val"

3514 local Keyword =
3515   K ( 'Keyword' ,
3516       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3517       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3518       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3519       + "virtual" + "when" + "while" + "with" )
3520   + K ( 'Keyword.Constant' , P "true" + "false" )
3521   + K ( 'Keyword.Governing', governing_keyword )

3522 local EndKeyword
3523   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3524   + EscapeMath + -1

```

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

```

3525 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3526                   - ( 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`.

```

3527 local Identifier = K ( 'Identifier.Internal' , identifier )

```

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

```

3528 local ocaml_char =
3529   P "'" *
3530   (
3531     ( 1 - S "'\\\" )
3532     + "\\\"
3533     * ( S "\\ntbr \"\"
3534         + digit * digit * digit
3535         + P "x" * ( digit + R "af" + R "AF" )
3536         * ( digit + R "af" + R "AF" )
3537         * ( digit + R "af" + R "AF" )
3538         + P "o" * R "03" * R "07" * R "07" )
3539   )
3540   * "'"
3541 local Char =
3542   K ( 'String.Short.Internal', ocaml_char )

```

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

```

3543 local TypeParameter =
3544   K ( 'TypeParameter' ,
3545       "'" * 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.

```

3546   local DotNotation =
3547       (
3548           K ( 'Name.Module' , cap_identifier )
3549           * Q "."
3550           * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3551           +
3552           Identifier
3553           * Q "."
3554           * K ( 'Name.Field' , identifier )
3555       )
3556       * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

```

## The records

```

3557   local expression_for_fields_type =
3558       P { "E" ,
3559           E = (   "{" * V "F" * "}"
3560                 + "(" * V "F" * ")"
3561                 + TypeParameter
3562                 + ( 1 - S "{ } ( [ ] \r ; " ) ) ^ 0 ,
3563           F = (   "{" * V "F" * "}"
3564                 + "(" * V "F" * ")"
3565                 + ( 1 - S "{ } ( [ ] \r \" \" ) + TypeParameter ) ^ 0
3566       }

```

```

3567   local expression_for_fields_value =
3568       P { "E" ,
3569           E = (   "{" * V "F" * "}"
3570                 + "(" * V "F" * ")"
3571                 + "[" * V "F" * "]"
3572                 + ocaml_string + ocaml_char
3573                 + ( 1 - S "{ } ( [ ] ; " ) ) ^ 0 ,
3574           F = (   "{" * V "F" * "}"
3575                 + "(" * V "F" * ")"
3576                 + "[" * V "F" * "]"
3577                 + ocaml_string + ocaml_char
3578                 + ( 1 - S "{ } ( [ ] \" \" ) ) ^ 0
3579       }

```

```

3580   local OneFieldDefinition =
3581       ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3582       * K ( 'Name.Field' , identifier ) * SkipSpace
3583       * Q ":" * SkipSpace
3584       * K ( 'TypeExpression' , expression_for_fields_type )
3585       * SkipSpace

```

```

3586   local OneField =
3587       K ( 'Name.Field' , identifier ) * SkipSpace
3588       * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3589       * ( C ( expression_for_fields_value ) / ParseAgain )
3590       * SkipSpace

```

## The records.

```

3591   local RecordVal =
3592       Q "{" * SkipSpace
3593       *
3594       (

```

```

3595     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3596   ) ^-1
3597   *
3598   (
3599     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3600   )
3601   * SkipSpace
3602   * Q ";" ^ -1
3603   * SkipSpace
3604   * Comment ^ -1
3605   * SkipSpace
3606   * Q "}"
3607   local RecordType =
3608     Q "{" * SkipSpace
3609     *
3610     (
3611       OneFieldDefinition
3612       * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3613     )
3614     * SkipSpace
3615     * Q ";" ^ -1
3616     * SkipSpace
3617     * Comment ^ -1
3618     * SkipSpace
3619     * Q "}"
3620   local Record = RecordType + RecordVal

```

```

3621   local Operator =
3622     P "||" *

```

Don't use \hspace instead of \kern!

```

3623   Lc([{\PitonStyle{Operator}}{\kern0.1em/\kern-0.2em/\kern0.1em}}]])
3624   +
3625   K ( 'Operator' ,
3626     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3627     "/" + "*" + ";" + "->" + "+." + "-." + "*." + "/" +
3628     + S "-~+/*%=<>&@|" )

```

```

3629   local Builtin =
3630     K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

3631   local Exception =
3632     K ( 'Exception' ,
3633       P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3634       "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3635       "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

```

```

3636   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).

```

3637   local pattern_part =
3638     ( 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).

```

3639   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.

```
3640      ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3641      *
```

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

```
3642      (
3643          K ( 'Identifier.Internal' , identifier )
3644      +
3645          Q "(" * SkipSpace
3646          * ( C ( pattern_part ) / ParseAgain )
3647          * SkipSpace
```

Of course, the specification of type is optional.

```
3648      * ( Q ":" * #(1- P"=")
3649          * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3650      ) ^ -1
3651      * Q ")"
3652  )
```

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

```
3653  local DefFunction =
3654      K ( 'Keyword.Governing' , "let open" )
3655      * Space
3656      * K ( 'Name.Module' , cap_identifier )
3657      +
3658      K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3659      * Space
3660      * K ( 'Name.Function.Internal' , identifier )
3661      * Space
3662      * (
```

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).

```
3663      Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3664      +
3665      Argument * ( SkipSpace * Argument ) ^ 0
3666      * (
3667          SkipSpace
3668          * Q ":" * # ( 1 - P "=" )
3669          * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3670      ) ^ -1
3671  )
```

## DefModule

```
3672  local DefModule =
3673      K ( 'Keyword.Governing' , "module" ) * Space
3674      *
3675      (
3676          K ( 'Keyword.Governing' , "type" ) * Space
3677          * K ( 'Name.Type' , cap_identifier )
3678      +
3679          K ( 'Name.Module' , cap_identifier ) * SkipSpace
3680          *
3681          (
3682              Q "(" * SkipSpace
3683              * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3684              * Q ":" * # ( 1 - P "=" ) * SkipSpace
3685              * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3686              *
3687              (
3688                  Q "," * SkipSpace
3689                  * K ( 'Name.Module' , cap_identifier ) * SkipSpace
```

```

3690         * Q ":" * # ( 1 - P "=" ) * SkipSpace
3691         * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3692     ) ^ 0
3693     * Q ")"
3694 ) ^ -1
3695 *
3696 (
3697     Q "=" * SkipSpace
3698     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3699     * Q "("
3700     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3701     *
3702     (
3703         Q ",",
3704         *
3705         K ( 'Name.Module' , cap_identifier ) * SkipSpace
3706     ) ^ 0
3707     * Q ")"
3708 ) ^ -1
3709 )
3710 +
3711 K ( 'Keyword.Governing' , P "include" + "open" )
3712 * Space
3713 * K ( 'Name.Module' , cap_identifier )

```

## DefType

```

3714 local DefType =
3715     K ( 'Keyword.Governing' , "type" )
3716     * Space
3717     * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3718     * SkipSpace
3719     * ( Q "+=" + Q "=" )
3720     * SkipSpace
3721     * (
3722         RecordType
3723         +

```

The following lines are a suggestion of Y. Salmon.

```

3724     WithStyle
3725     (
3726         'TypeExpression' ,
3727         (
3728             (
3729                 EOL
3730                 + comment
3731                 + Q ( 1
3732                     - P ";;"
3733                     - P "type"
3734                     - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3735                 )
3736             ) ^ 0
3737             *
3738             (
3739                 # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3740                 + Q ";;"
3741                 + -1
3742             )
3743         )
3744     )
3745 )

3746 local prompt =
3747     Q "utop[" * digit^1 * Q "> "

```

```

3748 local start_of_line = P(function(subject, position)
3749   if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3750     return position
3751   end
3752   return nil
3753 end)
3754 local Prompt = #start_of_line * K( 'Prompt', prompt )
3755 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3756               * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3757               * (K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

## The main LPEG for the language OCaml

```

3758 local Main =
3759   space ^ 0 * EOL
3760   + Space
3761   + Tab
3762   + Escape + EscapeMath
3763   + Beamer
3764   + DetectedCommands
3765   + TypeParameter
3766   + String + QuotedString + Char
3767   + Comment
3768   + Prompt + Answer

```

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

```

3769   + Q "~" * Identifier * ( Q ":" ) ^ -1
3770   + Q ":" * # (1 - P ":") * SkipSpace
3771     * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3772   + Exception
3773   + DefType
3774   + DefFunction
3775   + DefModule
3776   + Record
3777   + Keyword * EndKeyword
3778   + OperatorWord * EndKeyword
3779   + Builtin * EndKeyword
3780   + DotNotation * EndKeyword
3781   + Constructor
3782   + Identifier
3783   + Punct
3784   + Delim -- Delim is before Operator for a correct analysis of [| et |]
3785   + Operator
3786   + Number
3787   + Word

```

Here, we must not put local, of course.

```

3788 LPEG1.ocaml = Main ^ 0

```

```

3789 LPEG2.ocaml =
3790   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).

```

3791   (
3792     (
3793       P ":"
3794       +
3795       (
3796         ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3797         * Identifier

```

```

3798         * SkipSpace
3799         * Q ":"
3800     )
3801 )
3802     * # ( 1 - S "!=" )
3803     * SkipSpace
3804     * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 ) * -1
3805 )
3806 +
3807 (
3808     ( space ^ 0 * "\r" ) ^ -1
3809     * Lc [ [ \@_begin_line: ] ]
3810     * LeadingSpace ^ 0
3811     * ( ( space * Lc [ [ \@_trailing_space: ] ] ) ^ 1 * -1
3812         + space ^ 0 * EOL
3813         + Main
3814     ) ^ 0
3815     * -1
3816     * Lc [ [ \@_end_line: ] ]
3817 )
3818 )

```

End of the Lua scope for the language OCaml.

```

3819 end

```

### 3.6 The language C

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

```

3820 --c C c++ C++
3821 do
3822     local Delim = Q ( S "{[()]}" )
3823     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.

```

3824     local identifier = letter * alphanum ^ 0
3825
3826     local Operator =
3827         K ( 'Operator' ,
3828             P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3829             + S "-~/*%=<>&.@|!" )
3830
3831     local Keyword =
3832         K ( 'Keyword' ,
3833             P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3834             "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3835             "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3836             "register" + "restricted" + "return" + "static" + "static_assert" +
3837             "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3838             "union" + "using" + "virtual" + "volatile" + "while"
3839         )
3840         + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3841
3842     local Builtin =
3843         K ( 'Name.Builtin' ,
3844             P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3845
3846     local Type =

```

```

3847 K ( 'Name.Type' ,
3848     P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3849     "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3850     "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3851     "void" + "wchar_t" ) * Q "*" ^ 0
3852
3853 local DefFunction =
3854     Type
3855     * Space
3856     * Q "*" ^ -1
3857     * K ( 'Name.Function.Internal' , identifier )
3858     * SkipSpace
3859     * # 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:`

```

3860 local DefClass =
3861     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).

```

3862 local Character =
3863     K ( 'String.Short' ,
3864         P [['\']] + P "'" * ( 1 - P "'" ) ^ 0 * P "'" )

```

## The strings of C

```

3865 String =
3866     WithStyle ( 'String.Long.Internal' ,
3867         Q "\""
3868         * ( SpaceInString
3869             + K ( 'String.Interpol' ,
3870                 "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3871             )
3872         + Q ( ( P "\\\"" + 1 - S " \" " ) ^ 1 )
3873         ) ^ 0
3874         * Q "\""
3875     )

```

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

```

3876 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
3877 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end
3878
3879 DetectedCommands =
3880     Compute_DetectedCommands ( 'c' , braces )
3881     + Compute_RawDetectedCommands ( 'c' , braces )
3882
3883 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

## The directives of the preprocessor

```

3882 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.

```

3883 local Comment =
3884   WithStyle ( 'Comment.Internal' ,
3885     Q "/" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3886     * ( EOL + -1 )
3887
3888 local LongComment =
3889   WithStyle ( 'Comment.Internal' ,
3890     Q "/*"
3891     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3892     * Q "*/"
3893     ) -- $

```

## The main LPEG for the language C

```

3894 local EndKeyword
3895   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3896     EscapeMath + -1

```

First, the main loop :

```

3897 local Main =
3898   space ^ 0 * EOL
3899   + Space
3900   + Tab
3901   + Escape + EscapeMath
3902   + CommentLaTeX
3903   + Beamer
3904   + DetectedCommands
3905   + Preproc
3906   + Comment + LongComment
3907   + Delim
3908   + Operator
3909   + Character
3910   + String
3911   + Punct
3912   + DefFunction
3913   + DefClass
3914   + Type * ( Q "*" ^ -1 + EndKeyword )
3915   + Keyword * EndKeyword
3916   + Builtin * EndKeyword
3917   + Identifier
3918   + Number
3919   + Word

```

Here, we must not put local, of course.

```

3920 LPEG1.c = Main ^ 0

```

We recall that each line in the C code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>7</sup>.

```

3921 LPEG2.c =
3922   Ct (
3923     ( space ^ 0 * P "\r" ) ^ -1
3924     * Lc [[ \@@_begin_line: ]]
3925     * LeadingSpace ^ 0
3926     * ( space ^ 1 * -1 + Main ) ^ 0
3927     * -1
3928     * Lc [[ \@@_end_line: ]]
3929   )

```

---

<sup>7</sup>Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

End of the Lua scope for the language C.

```
3930 end
```

### 3.7 The language SQL

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

```
3931 --sql SQL
3932 do

3933     local LuaKeyword
3934     function LuaKeyword ( name ) return
3935         Lc [[ {\PitonStyle{Keyword}{ } }
3936         * Q ( Cmt (
3937             C ( letter * alphanum ^ 0 ) ,
3938             function ( _ , _ , a ) return a : upper ( ) == name end
3939         )
3940     )
3941     * Lc "}}"
3942 end
```

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

```
3943     local identifier =
3944         letter * ( alphanum + "-" ) ^ 0
3945         + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"
3946     local Operator =
3947         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).

```
3948     local Set
3949     function Set ( list )
3950         local set = { }
3951         for _ , l in ipairs ( list ) do set[l] = true end
3952         return set
3953     end
```

We now use the previous function `Set` to create the “sets” `set_keywords` and `set_builtin`. That list of keywords comes from [https://sqlite.org/lang\\_keywords.html](https://sqlite.org/lang_keywords.html).

```
3954     local set_keywords = Set
3955     {
3956         "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3957         "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3958         "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3959         "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3960         "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3961         "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3962         "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3963         "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3964         "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3965         "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3966         "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3967         "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
```

```

3968     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3969     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3970     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3971     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3972     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3973     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3974     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3975     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3976 }
3977 local set_builtins = Set
3978 {
3979     "AVG" , "COUNT" , "CHAR_LENGTH" , "CONCAT" , "CURDATE" , "CURRENT_DATE" ,
3980     "DATE_FORMAT" , "DAY" , "LOWER" , "LTRIM" , "MAX" , "MIN" , "MONTH" , "NOW" ,
3981     "RANK" , "ROUND" , "RTRIM" , "SUBSTRING" , "SUM" , "UPPER" , "YEAR"
3982 }

```

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.

```

3983 local Identifier =
3984 C ( identifier ) /
3985 (
3986     function ( s )
3987         if set_keywords [ s : upper ( ) ] then return

```

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

```

3988         { [{\PitonStyle{Keyword}{}}] } ,
3989         { luatexbase.catcodetables.other , s } ,
3990         { "}" } }
3991     else
3992         if set_builtins [ s : upper ( ) ] then return
3993         { [{\PitonStyle{Name.Builtin}{}}] } ,
3994         { luatexbase.catcodetables.other , s } ,
3995         { "}" } }
3996     else return
3997     { [{\PitonStyle{Name.Field}{}}] } ,
3998     { luatexbase.catcodetables.other , s } ,
3999     { "}" } }
4000     end
4001 end
4002 end
4003 )

```

## The strings of SQL

```

4004 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.

```

4005 local braces = Compute_braces ( "'" * ( 1 - P "'" ) ^ 1 * "'" )
4006 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
4007 DetectedCommands =
4008     Compute_DetectedCommands ( 'sql' , braces )
4009     + Compute_RawDetectedCommands ( 'sql' , braces )
4010 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.

```

4011 local Comment =
4012     WithStyle ( 'Comment.Internal' ,
4013         Q "--" -- syntax of SQL92
4014         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
4015     * ( EOL + -1 )
4016
4017 local LongComment =
4018     WithStyle ( 'Comment.Internal' ,
4019         Q "/*"
4020         * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
4021         * Q "*/"
4022     ) -- $

```

**The main LPEG for the language SQL**

```

4023 local EndKeyword
4024     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
4025     EscapeMath + -1
4026
4027 local TableField =
4028     K ( 'Name.Table' , identifier )
4029     * Q "."
4030     * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
4031
4032 local OneField =
4033     (
4034         Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
4035         +
4036         K ( 'Name.Table' , identifier )
4037         * Q "."
4038         * K ( 'Name.Field' , identifier )
4039         +
4040         K ( 'Name.Field' , identifier )
4041     )
4042     * (
4043         Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
4044     ) ^ -1
4045     * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
4046
4047 local OneTable =
4048     K ( 'Name.Table' , identifier )
4049     * (
4050         Space
4051         * LuaKeyword "AS"
4052         * Space
4053         * K ( 'Name.Table' , identifier )
4054     ) ^ -1
4055
4056 local WeCatchTableNames =
4057     LuaKeyword "FROM"
4058     * ( Space + EOL )
4059     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
4060     + (
4061         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
4062         + LuaKeyword "TABLE"
4063     )
4064     * ( Space + EOL ) * OneTable
4065
4066 local EndKeyword
4067     = Space + Punct + Delim + EOL + Beamer
4068     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

4067 local Main =
4068     space ^ 0 * EOL
4069     + Space
4070     + Tab
4071     + Escape + EscapeMath
4072     + CommentLaTeX
4073     + Beamer
4074     + DetectedCommands
4075     + Comment + LongComment
4076     + Delim
4077     + Operator
4078     + String
4079     + Punct
4080     + WeCatchTableNames
4081     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
4082     + Number
4083     + Word

```

Here, we must not put local, of course.

```

4084 LPEG1.sql = Main ^ 0

```

We recall that each line in the code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`<sup>8</sup>.

```

4085 LPEG2.sql =
4086     Ct (
4087         ( space ^ 0 * "\r" ) ^ -1
4088         * Lc [[ \@@_begin_line: ]]
4089         * LeadingSpace ^ 0
4090         * ( space ^ 1 * -1 + Main ) ^ 0
4091         * -1
4092         * Lc [[ \@@_end_line: ]]
4093     )

```

End of the Lua scope for the language SQL.

```

4094 end

```

### 3.8 The language “Minimal”

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

```

4095 --minimal Minimal
4096 do
4097     local Punct = Q ( S " , ; ! \ " )
4098
4099     local Comment =
4100         WithStyle ( 'Comment.Internal' ,
4101             Q "#"
4102             * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4103         )
4104     * ( EOL + -1 )
4105
4106     local String =
4107         WithStyle ( 'String.Short.Internal' ,
4108             Q "\""
4109             * ( SpaceInString
4110                 + Q ( ( P [[\]] + 1 - S " \"" ) ^ 1 )

```

---

<sup>8</sup>Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

```

4111         ) ^ 0
4112         * Q "\""
4113     )

```

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

```

4114     local braces = Compute_braces ( P "\"" * ( P "\\\"" + 1 - P "\"" ) ^ 1 * "\"" )
4115
4116     if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
4117
4118     DetectedCommands =
4119         Compute_DetectedCommands ( 'minimal' , braces )
4120         + Compute_RawDetectedCommands ( 'minimal' , braces )
4121
4122     LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
4123
4124     local identifier = letter * alphanum ^ 0
4125
4126     local Identifier = K ( 'Identifier.Internal' , identifier )
4127
4128     local Delim = Q ( S "{[()]}")
4129
4130     local Main =
4131         space ^ 0 * EOL
4132         + Space
4133         + Tab
4134         + Escape + EscapeMath
4135         + CommentLaTeX
4136         + Beamer
4137         + DetectedCommands
4138         + Comment
4139         + Delim
4140         + String
4141         + Punct
4142         + Identifier
4143         + Number
4144         + Word

```

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

```

4145     LPEG1.minimal = Main ^ 0
4146
4147     LPEG2.minimal =
4148         Ct (
4149             ( space ^ 0 * "\r" ) ^ -1
4150             * Lc [[ \@@_begin_line: ]]
4151             * LeadingSpace ^ 0
4152             * ( space ^ 1 * -1 + Main ) ^ 0
4153             * -1
4154             * Lc [[ \@@_end_line: ]]
4155         )

```

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

```

4156 end

```

### 3.9 The language “Verbatim”

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

```

4157 --verbatim Verbatim
4158 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”).

```

4159 local braces =
4160     P { "E" ,
4161         E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4162     }
4163
4164 if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
4165
4166 DetectedCommands =
4167     Compute_DetectedCommands ( 'verbatim' , braces )
4168     + Compute_RawDetectedCommands ( 'verbatim' , braces )
4169
4170 LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

4171 local lpeg_central = 1 - S " \\r"
4172 if piton.begin_escape then
4173     lpeg_central = lpeg_central - piton.begin_escape
4174 end
4175 if piton.begin_escape_math then
4176     lpeg_central = lpeg_central - piton.begin_escape_math
4177 end
4178 local Word = Q ( lpeg_central ^ 1 )
4179
4180 local Main =
4181     space ^ 0 * EOL
4182     + Space
4183     + Tab
4184     + Escape + EscapeMath
4185     + Beamer
4186     + DetectedCommands
4187     + Q [[\]]
4188     + Word

```

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

```

4189 LPEG1.verbatim = Main ^ 0
4190
4191 LPEG2.verbatim =
4192     Ct (
4193         ( space ^ 0 * "\r" ) ^ -1
4194         * Lc [[ \@@_begin_line: ]]
4195         * LeadingSpace ^ 0
4196         * ( space ^ 1 * -1 + Main ) ^ 0
4197         * -1
4198         * Lc [[ \@@_end_line: ]]
4199     )

```

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

```

4200 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).

```

4201 --EXPL expl
4202 do
4203     local Comment =
4204         WithStyle
4205         ( 'Comment.Internal' ,
4206         Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $

```

```

4207     )
4208     * ( EOL + -1 )

```

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

```

4209     local analyze_cs
4210     function analyze_cs ( s )
4211         local i = s : find ( ":" )
4212         if i then

```

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

```

4213         local name = s : sub ( 2 , i - 1 )
4214         local parts = name : explode ( "_" )
4215         local module = parts[1]
4216         if module == "" then module = parts[3] end

```

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

```

4217         return
4218         { [[{\OptionalLocalPitonStyle{Module.}} .. module .. "}{ " } ,
4219           { luatexbase.catcodetables.other , s } ,
4220           { "}" } }
4221     else
4222         local p = s : sub ( 1 , 3 )
4223         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`.

```

4224         local scope = s : sub(2,2)
4225         local parts = s : explode ( "_" )
4226         local module = parts[2]
4227         if module == "" then module = parts[3] end
4228         local type = parts[#parts]
4229         return
4230         { [[{\OptionalLocalPitonStyle{Scope.}} .. scope .. "}{ " } ,
4231           { [[{\OptionalLocalPitonStyle{Module.}} .. module .. "}{ " } ,
4232           { [[{\OptionalLocalPitonStyle{Type.}} .. type .. "}{ " } ,
4233           { luatexbase.catcodetables.other , s } ,
4234           { "}}}}}} " }
4235     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.

```

4236         return { luatexbase.catcodetables.other , s }
4237     end
4238 end
4239 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`).

```

4240     local braces =
4241     P { "E" ,
4242       E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4243     }
4244
4245     if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
4246
4247     DetectedCommands =
4248     Compute_DetectedCommands ( 'expl' , braces )
4249     + Compute_RawDetectedCommands ( 'expl' , braces )
4250
4251     LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
4252
4253     local control_sequence = P "\\\" * ( R "Az" + "_" + ":" + "@" ) ^ 1
4254     local ControlSequence = C ( control_sequence ) / analyze_cs

```

```

4254 local def_function
4255   = P [[\cs_]]
4256     * ( P "set" + "new" )
4257     * ( P "_protected" ) ^ -1
4258     * P ":N" * ( P "p" ) ^ -1 * "n"

4259 local DefFunction =
4260   C ( def_function ) / analyze_cs
4261   * Space
4262   * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
4263   * ControlSequence -- Q ( ControlSequence ) ?
4264   * Lc "}"

4265 local Word = Q ( ( 1 - S " \r" ) ^ 1 )

4266
4267 local Main =
4268   space ^ 0 * EOL
4269   + Space
4270   + Tab
4271   + Escape + EscapeMath
4272   + Beamer
4273   + Comment
4274   + DetectedCommands
4275   + DefFunction
4276   + ControlSequence
4277   + Word

```

Here, we must not put local, of course.

```

4278 LPEG1.expl = Main ^ 0
4279
4280 LPEG2.expl =
4281   Ct (
4282     ( space ^ 0 * " \r" ) ^ -1
4283     * Lc [[ \@@_begin_line: ] ]
4284     * LeadingSpace ^ 0
4285     * ( space ^ 1 * -1 + Main ) ^ 0
4286     * -1
4287     * Lc [[ \@@_end_line: ] ]
4288   )

```

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

```

4289 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.

```

4290 function piton.Parse ( language , code )

```

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

```

4291   piton.language = language
4292   local t = LPEG2[language] : match ( code )
4293   if not t then
4294     sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ] ]
4295     return -- to exit in force the function
4296   end
4297   local left_stack = {}
4298   local right_stack = {}

```

```

4299   for _ , one_item in ipairs ( t ) do
4300       if one_item == "EOL" then
4301           for i = #right_stack, 1, -1 do
4302               tex.sprint ( right_stack[i] )
4303           end

```

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

```

4304       sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
4305       tex.sprint ( table.concat ( left_stack ) )
4306   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}`.

```

4307       if one_item[1] == "Open" then
4308           tex.sprint ( one_item[2] )
4309           table.insert ( left_stack , one_item[2] )
4310           table.insert ( right_stack , one_item[3] )
4311       else
4312           if one_item[1] == "Close" then
4313               tex.sprint ( right_stack[#right_stack] )
4314               left_stack[#left_stack] = nil
4315               right_stack[#right_stack] = nil
4316           else
4317               tex.tprint ( one_item )
4318           end
4319       end
4320   end
4321 end
4322 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).

```

4323 local my_file_lines
4324 function my_file_lines ( filename )
4325     local f = io.open ( filename , 'rb' )
4326     local s = f : read ( '*a' )
4327     f : close ( )

```

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

```

4328     return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
4329 end

```

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

```

4330 function piton.ReadFile ( name , first_line , last_line )
4331     local s = ''
4332     local i = 0
4333     for line in my_file_lines ( name ) do
4334         i = i + 1
4335         if i >= first_line then
4336             s = s .. '\r' .. line
4337         end
4338         if i >= last_line then break end
4339     end

```

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

```

4340   if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
4341       s = s : sub ( 5 , -1 )
4342   end

4343   sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4344   tex.sprint ( luatexbase.catcodetables.other , s )
4345   sprintL3 ( "]" )
4346 end

4347 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
4348     local s
4349     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4350     piton.GobbleParse ( lang , n , splittable , s )
4351 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 `#`.

```

4352 function piton.ParseBis ( lang , code )
4353     return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
4354 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.

```

4355 function piton.ParseTer ( lang , code )

```

Be careful: we have to write `[[\@@_breakable_space: ]]` with a space after the name of the LaTeX command `\@@_breakable_space:`.

```

4356     return piton.Parse
4357         (
4358             lang ,
4359             code : gsub ( [[\@@_breakable_space: ]] , ' ' )
4360         )
4361 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.

```

4362 local AutoGobbleLPEG =
4363     ( (
4364         P " " ^ 0 * "\r"
4365         +
4366         Ct ( C " " ^ 0 ) / table.getn
4367         * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
4368     ) ^ 0
4369     * ( Ct ( C " " ^ 0 ) / table.getn
4370         * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4371 ) / math.min

```



The following LPEG is similar but works with the tabulations.

```

4372 local TabsAutoGobbleLPEG =
4373   (
4374     (
4375       P "\t" ^ 0 * "\r"
4376       +
4377       Ct ( C "\t" ^ 0 ) / table.getn
4378       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
4379     ) ^ 0
4380     * ( Ct ( C "\t" ^ 0 ) / table.getn
4381         * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4382   ) / 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).

```

4383 local EnvGobbleLPEG =
4384   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
4385   * 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.

```

4386 function piton.Gobble ( n , code )
4387   if n == 0 then return
4388     code
4389   else
4390     if n == -1 then
4391       n = AutoGobbleLPEG : match ( code )

```

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

```

4392     if tonumber(n) then else n = 0 end
4393   else
4394     if n == -2 then
4395       n = EnvGobbleLPEG : match ( code )
4396     else
4397       if n == -3 then
4398         n = TabsAutoGobbleLPEG : match ( code )
4399         if tonumber(n) then else n = 0 end
4400       end
4401     end
4402   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...

```

4403   if n == 0 then return
4404     code
4405   else return

```

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

```

4406   ( Ct (
4407     ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4408     * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4409     ) ^ 0 )
4410     / table.concat
4411   ) : match ( code )
4412 end
4413 end
4414 end

```

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

`splittable` is the value of `\l_@@_splittable_int`.

```

4415 function piton.GobbleParse ( lang , n , splittable , code )

```

```

4416 piton.ComputeLinesStatus ( code , splittable )
4417 piton.last_code = piton.Gobble ( n , code )
4418 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`.

```

4419 piton.CountLines ( piton.last_code )
4420 piton.Parse ( lang , piton.last_code )
4421 piton.join_and_write ( )
4422 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`.

```

4423 function piton.join_and_write ( )
4424   if piton.join ~= '' then
4425     if not piton.join_files [ piton.join ] then
4426       piton.join_files [ piton.join ] = piton.get_last_code ( )
4427     else
4428       if piton.join_separation == '' then
4429         piton.join_files [ piton.join ] =
4430           piton.join_files [ piton.join ]
4431           .. "\r\n"
4432           .. piton.get_last_code ( )
4433       else
4434         piton.join_files [ piton.join ] =
4435           piton.join_files [ piton.join ]
4436           .. "\r\n"
4437           .. ( piton.join_separation : gsub ( '##' , '#' ) )
4438           .. "\r\n"
4439           .. piton.get_last_code ( )
4440       end
4441     end
4442   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.

```

4443   if piton.write ~= '' then

```

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

```

4444     local file_name = ''
4445     if piton.path_write == '' then
4446       file_name = piton.write
4447     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.

```

4448       local attr = lfs.attributes ( piton.path_write )
4449       if attr and attr.mode == "directory" then
4450         file_name = piton.path_write .. "/" .. piton.write
4451       else

```

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

```

4452         sprintL3 [[ \@_error_or_warning:n { InexistentDirectory } ]]
4453       end
4454     end
4455     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`).

```

4456     if not piton.write_files [ file_name ] then
4457         piton.write_files [ file_name ] = piton.get_last_code ( )
4458     else
4459         piton.write_files [ file_name ] =
4460         piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4461     end
4462 end
4463 end
4464 end

```

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

```

4465 function piton.GobbleParseNoPrint ( lang , n , code )
4466     piton.last_code = piton.Gobble ( n , code )
4467     piton.last_language = lang
4468     piton.join_and_write ( )
4469 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).

```

4470 function piton.GobbleSplitParse ( lang , n , splittable , code )
4471     local chunks
4472     chunks =
4473     (
4474         Ct (
4475             (
4476                 P " " ^ 0 * "\r"
4477                 +
4478                 C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4479                     - ( P " " ^ 0 * ( P "\r" + -1 ) )
4480                 ) ^ 1
4481             )
4482         ) ^ 0
4483     )
4484     ) : match ( piton.Gobble ( n , code ) )
4485     sprintL3 [[ \begingroup ]]
4486     sprintL3
4487     (
4488         [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
4489         .. "language = " .. lang .. ","
4490         .. "splittable = " .. splittable .. "]"
4491     )
4492     for k , v in pairs ( chunks ) do
4493         if k > 1 then
4494             sprintL3 ( [[ \l_@@_split_separation_tl ]] )
4495         end
4496         tex.print
4497         (
4498             [[\begin{}} .. piton.env_used_by_split .. "}\r"
4499             .. v
4500             .. [[\end{}} .. piton.env_used_by_split .. "}\r"
4501         )
4502     end
4503     sprintL3 [[ \endgroup ]]
4504 end

```

```

4505 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4506   local s
4507   s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4508   piton.GobbleSplitParse ( lang , n , splittable , s )
4509 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.

```

4510 piton.string_between_chunks =
4511 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4512 .. [[ \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.

```

4513 function piton.get_last_code ( )
4514   return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4515   : gsub ( '\r\n?' , '\n' )
4516 end

```

### 3.14 To count the number of lines

```

4517 local CountBeamerEnvironments
4518 function CountBeamerEnvironments ( code ) return
4519   (
4520     Ct (
4521       (
4522         P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4523         +
4524         ( 1 - P "\r" ) ^ 0 * "\r"
4525       ) ^ 0
4526       * ( 1 - P "\r" ) ^ 0
4527       * -1
4528     ) / table.getn
4529   ) : match ( code )
4530 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.

```

4531 function piton.CountLines ( code )
4532   local count
4533   count =
4534     ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4535       *
4536       (
4537         space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4538         + space ^ 0
4539       ) ^ -1
4540       * -1
4541     ) / table.getn
4542   ) : match ( code )
4543   if piton.beamer then
4544     count = count - 2 * CountBeamerEnvironments ( code )
4545   end
4546   sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4547 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).

```
4548 function piton.CountNonEmptyLines ( code )
4549     local count = 0
```

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

```
4550     count =
4551         ( Ct ( ( P " " ^ 0 * "\r"
4552             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4553             * ( 1 - P "\r" ) ^ 0
4554             * -1
4555             ) / table.getn
4556         ) : match ( code )
4557     count = count + 1
4558     if piton.beamer then
4559         count = count - 2 * CountBeamerEnvironments ( code )
4560     end
4561     sprintL3
4562         ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4563 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.

```
4564 function piton.ComputeRange ( s , t , file_name )
4565     local first_line = -1
4566     local count = 0
4567     local last_found = false
4568     for line in io.lines ( file_name ) do
4569         if first_line == -1 then
4570             if line : sub ( 1 , #s ) == s then
4571                 first_line = count
4572             end
4573         else
4574             if line : sub ( 1 , #t ) == t then
4575                 last_found = true
4576                 break
4577             end
4578         end
4579         count = count + 1
4580     end
4581     if first_line == -1 then
4582         sprintL3 [[ \@@_error_or_warning:n { begin-marker-not-found } ]]
4583     else
4584         if not last_found then
4585             sprintL3 [[ \@@_error_or_warning:n { end-marker-not-found } ]]
4586         end
4587     end
4588     sprintL3 (
4589         [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 }'
4590         .. [[ \global \l_@@_last_line_int = ]] .. count )
4591 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`.

```
4592 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).

```
4593   local lpeg_line_beamer
4594   if piton.beamer then
4595     lpeg_line_beamer =
4596       space ^ 0
4597       * P [[\begin{]] * beamerEnvironments * "]"
4598       * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4599       +
4600       space ^ 0
4601       * P [[\end{]] * beamerEnvironments * "]"
4602   else
4603     lpeg_line_beamer = P ( false )
4604   end
4605   local lpeg_empty_lines =
4606     Ct (
4607       ( lpeg_line_beamer * "\r"
4608         +
4609         P " " ^ 0 * "\r" * Cc ( 0 )
4610         +
4611         ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4612       ) ^ 0
4613       *
4614       ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4615     )
4616     * -1
4617   local lpeg_all_lines =
4618     Ct (
4619       ( lpeg_line_beamer * "\r"
4620         +
4621         ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4622       ) ^ 0
4623       *
4624       ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4625     )
4626     * -1
```

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

```
4627   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`).

```
4628   local lines_status
4629   local s = splittable
4630   if splittable < 0 then s = - splittable end
```

```

4631 if splittable > 0 then
4632   lines_status = lpeg_all_lines : match ( code )
4633 else

```

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

```

4634   lines_status = lpeg_empty_lines : match ( code )
4635   for i , x in ipairs ( lines_status ) do
4636     if x == 0 then
4637       for j = 1 , s - 1 do
4638         if i + j > #lines_status then break end
4639         if lines_status[i+j] == 0 then break end
4640         lines_status[i+j] = 2
4641       end
4642       for j = 1 , s - 1 do
4643         if i - j == 1 then break end
4644         if lines_status[i-j-1] == 0 then break end
4645         lines_status[i-j-1] = 2
4646       end
4647     end
4648   end
4649 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.

```

4650   for j = 1 , s - 1 do
4651     if j > #lines_status then break end
4652     if lines_status[j] == 0 then break end
4653     lines_status[j] = 2
4654   end

```

Now, from the end of the code.

```

4655   for j = 1 , s - 1 do
4656     if #lines_status - j == 0 then break end
4657     if lines_status[#lines_status - j] == 0 then break end
4658     lines_status[#lines_status - j] = 2
4659   end

```

```

4660   piton.lines_status = lines_status
4661 end

4662 function piton.TranslateBeamerEnv ( code )
4663   local s
4664   s =
4665   (
4666     Ct (
4667       (
4668         space ^ 0
4669         * C (
4670           ( P "\\begin{" + "\\end{" )
4671           * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * "\r"
4672         )
4673         + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4674       ) ^ 0
4675       *
4676       (
4677         (
4678           space ^ 0
4679           * C (
4680             ( P "\\begin{" + "\\end{" )
4681             * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * -1
4682           )
4683           + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4684         ) ^ -1

```

```

4685         )
4686     ) ^ -1 / table.concat
4687 ) : match ( code )
4688 sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4689 tex.sprint ( luatexbase.catcodetables.other , s )
4690 sprintL3 ( "}" )
4691 end

```

### 3.16 To create new languages with the syntax of listings

```

4692 function piton.new_language ( lang , definition )
4693     lang = lang : lower ( )

4694     local alpha , digit = lpeg.alpha , lpeg.digit
4695     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.

```

4696     function add_to_letter ( c )
4697         if c ~= " " then table.insert ( extra_letters , c ) end
4698     end

```

For the digits, it's straitforward.

```

4699     function add_to_digit ( c )
4700         if c ~= " " then digit = digit + c end
4701     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 `}`).

```

4702     local other = S " :_@+*/<>!?.() []~^=#&\"'\\$" --
4703     local extra_others = { }
4704     function add_to_other ( c )
4705         if c ~= " " then

```

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

```

4706         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 `</....>`).

```

4707         other = other + P ( c )
4708     end
4709 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`.

```

4710     local def_table
4711     if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4712         def_table = {}
4713     else
4714         local strict_braces =
4715             P { "E" ,
4716                 E = ( "{" * V "F" * "}" + ( 1 - S "{ }" ) ) ^ 0 ,
4717                 F = ( "{" * V "F" * "}" + ( 1 - S "{" ) ) ^ 0
4718             }
4719         local cut_definition =
4720             P { "E" ,
4721                 E = Ct ( V "F" * ( " , " * V "F" ) ^ 0 ) ,
4722                 F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0

```



```

4723         * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4724     }
4725     def_table = cut_definition : match ( definition )
4726 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.).

```

4727 local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4728 local tex_arg = tex_braced_arg + C ( 1 )
4729 local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4730 local args_for_tag
4731   = tex_option_arg
4732     * space ^ 0
4733     * tex_arg
4734     * space ^ 0
4735     * tex_arg
4736 local args_for_morekeywords
4737   = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4738     * space ^ 0
4739     * tex_option_arg
4740     * space ^ 0
4741     * tex_arg
4742     * space ^ 0
4743     * ( tex_braced_arg + Cc ( nil ) )
4744 local args_for_moredelims
4745   = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4746     * args_for_morekeywords
4747 local args_for_morecomment
4748   = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4749     * space ^ 0
4750     * tex_option_arg
4751     * space ^ 0
4752     * 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`.

```

4753 local sensitive = true
4754 local style_tag , left_tag , right_tag
4755 for _ , x in ipairs ( def_table ) do
4756   if x[1] == "sensitive" then
4757     if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4758       sensitive = true
4759     else
4760       if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4761     end
4762   end
4763   if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4764   if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4765   if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4766   if x[1] == "tag" then
4767     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4768     style_tag = style_tag or [[\PitonStyle{Tag}]]
4769   end
4770 end

```

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

```

4771 local Number =
4772   K ( 'Number.Internal' ,
4773     ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0

```

```

4774         + digit ^ 0 * "." * digit ^ 1
4775         + digit ^ 1 )
4776     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4777     + digit ^ 1
4778 )
4779 local string_extra_letters = ""
4780 for _ , x in ipairs ( extra_letters ) do
4781     if not ( extra_others[x] ) then
4782         string_extra_letters = string_extra_letters .. x
4783     end
4784 end
4785 local letter = alpha + S ( string_extra_letters )
4786         + P "â" + "ä" + "ç" + "é" + "ê" + "ë" + "ï" + "î"
4787         + "ô" + "û" + "ü" + "Ë" + "Ê" + "Ç" + "É" + "È" + "Ë" + "Ê"
4788         + "Ï" + "Î" + "Ï" + "Û" + "Ü"
4789 local alphanum = letter + digit
4790 local identifier = letter * alphanum ^ 0
4791 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.

```

4792 local split_clist =
4793     P { "E" ,
4794         E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4795             * ( P "{" ) ^ 1
4796             * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4797             * ( P "}" ) ^ 1 * space ^ 0 ,
4798         F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4799     }

```

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

```

4800 local keyword_to_lpeg
4801 function keyword_to_lpeg ( name ) return
4802     Q ( Cmt (
4803         C ( identifier ) ,
4804         function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4805         end
4806     ) )
4807 )
4808 end
4809 local Keyword = P ( false )
4810 local PrefixedKeyword = P ( false )

```

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

```

4811 for _ , x in ipairs ( def_table )
4812 do if x[1] == "morekeywords"
4813     or x[1] == "otherkeywords"
4814     or x[1] == "moredirectives"
4815     or x[1] == "moretexcs"
4816 then
4817     local keywords = P ( false )
4818     local style = [[\PitonStyle{Keyword}]]
4819     if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
4820     style = tex_option_arg : match ( x[2] ) or style
4821     local n = tonumber ( style )
4822     if n then
4823         if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4824     end
4825     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4826         if x[1] == "moretexcs" then
4827             keywords = Q ( [[\]] .. word ) + keywords
4828         else
4829             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*.

```

4830         then keywords = Q ( word ) + keywords
4831         else keywords = keyword_to_lpeg ( word ) + keywords
4832     end
4833 end
4834 end
4835 Keyword = Keyword +
4836     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4837 end

```

Of course, the feature with the key `keywordsprefix` is designed for the languages TeX, LaTeX, et al. In that case, there is two kinds of keywords (= control sequences).

- those beginning with \ and a sequence of characters of catcode “letter”;
- those beginning by \ followed by one character of catcode “other”.

The following code addresses both cases. Of course, the LPEG pattern `letter` must catch only characters of catcode “letter”. That's why we have a key `alsoletter` to add new characters in that category (e.g. : when we want to format L3 code). However, the LPEG pattern is allowed to catch *more* than only the characters of catcode “other” in TeX.

```

4838     if x[1] == "keywordsprefix" then
4839         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4840         PrefixedKeyword = PrefixedKeyword
4841             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4842     end
4843 end

```

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

```

4844     local long_string = P ( false )
4845     local Long_string = P ( false )
4846     local LongString = P ( false )
4847     local central_pattern = P ( false )
4848     for _ , x in ipairs ( def_table ) do
4849         if x[1] == "morestring" then
4850             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4851             arg2 = arg2 or [[\PitonStyle{String.Long}]]
4852             if arg1 ~= "s" then
4853                 arg4 = arg3
4854             end
4855             central_pattern = 1 - S ( " \r" .. arg4 )
4856             if arg1 : match "b" then
4857                 central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4858             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...

```

4859             if arg1 : match "d" or arg1 == "m" then
4860                 central_pattern = P ( arg3 .. arg3 ) + central_pattern
4861             end
4862             if arg1 == "m"
4863             then prefix = B ( 1 - letter - ")" - "]" )
4864             else prefix = P ( true )
4865             end

```

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

```

4866         long_string = long_string +
4867             prefix
4868             * arg3
4869             * ( space + central_pattern ) ^ 0
4870             * arg4

```

Now a pattern *with captures*.

```

4871     local pattern =
4872         prefix
4873         * Q ( arg3 )
4874         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4875         * Q ( arg4 )

```

We will need Long\_string in the nested comments.

```

4876     Long_string = Long_string + pattern
4877     LongString = LongString +
4878         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4879         * pattern
4880         * Ct ( Cc "Close" )
4881     end
4882 end

```

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

```

4883     local braces = Compute_braces ( long_string )
4884     if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4885
4886     DetectedCommands =
4887         Compute_DetectedCommands ( lang , braces )
4888         + Compute_RawDetectedCommands ( lang , braces )
4889
4890     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*).

```

4891     local sComment = P ( false )

```

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

```

4892     local lComment = P ( false )

```

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

```

4893     local fComment = P ( false )
4894     local fComment = P ( false )
4895
4896     for _ , x in ipairs ( def_table ) do
4897         if x[1] == "morecomment" then
4898             local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4899             arg2 = arg2 or [[\PitonStyle{Comment}]]

```

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

```

4900         if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4901         if arg1 : match "l" or arg1 : match "f" then
4902             local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4903                 : match ( other_args )
4904             if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4905             if arg3 == [[\%]] then arg3 = "%" end -- mandatory
4906             local MyLPEG =
4907                 Ct ( Cc "Open"
4908                     * Cc ( "{" .. arg2 .. [[{\PitonSpaceSubstitute{}}] ]
4909                     * Cc "}" )
4910                 )
4911                 * Q ( arg3 )
4912                 * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $ noqa
4913                 * Ct ( Cc "Close" )
4914                 * ( EOL + -1 )
4915             if arg1 : match "l" then
4916                 lComment = lComment + MyLPEG

```

```

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

```

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

```

4955 if x[1] == "moredelim" then
4956     local arg1 , arg2 , arg3 , arg4 , arg5
4957     = args_for_moredelims : match ( x[2] )
4958     local MyFun = Q
4959     if arg1 == "*" or arg1 == "**" then
4960         function MyFun ( x )
4961             if x ~= '' then return
4962                 LPEG1[lang] : match ( x )
4963             end
4964         end
4965     end
4966     local left_delim
4967     if arg2 : match "i" then
4968         left_delim = P ( arg4 )
4969     else
4970         left_delim = Q ( arg4 )
4971     end
4972     if arg2 : match "l" then
4973         sComment = sComment +
4974             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4975             * left_delim
4976             * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4977             * Ct ( Cc "Close" )
4978             * ( EOL + -1 )

```

```

4979     end
4980     if arg2 : match "s" then
4981         local right_delim
4982         if arg2 : match "i" then
4983             right_delim = P ( arg5 )
4984         else
4985             right_delim = Q ( arg5 )
4986         end
4987         sComment = sComment +
4988             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4989             * left_delim
4990             * ( MyFun ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
4991             * right_delim
4992             * Ct ( Cc "Close" )
4993     end
4994 end
4995 end
4996
4997 local Delim = Q ( S "{[()]}")
4998 local Punct = Q ( S "=:;!\\" )
4999
5000 local EOL =
5001     P "\r"
5002     *
5003     (
5004         space ^ 0 * -1
5005         +
5006         Cc "EOL"
5007     )
5008     * ( ( fComment + lComment ) ^ 0 * LeadingSpace ^ 0 * # ( 1 - S "\r" ) ) ^ -1
5009
5010 local Main =
5011     space ^ 0 * EOL * ( fComment + lComment ) ^ 0
5012     + Space
5013     + Tab
5014     + Escape + EscapeMath
5015     + CommentLaTeX
5016     + Beamer
5017     + DetectedCommands
5018     + sComment
5019     + lComment

```

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

```

5018     + LongString
5019     + Delim
5020     + PrefixedKeyword
5021     + Keyword * ( -1 + # ( 1 - alphanum ) )
5022     + Punct
5023     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5024     + Number
5025     + 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.

```

5026     LPEG1[lang] = Main ^ 0

```

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

```

5027     LPEG2[lang] =
5028         Ct (
5029             ( space ^ 0 * P "\r" ) ^ -1
5030             * Lc [["@_begin_line: ]]
5031             * ( fComment + lComment ) ^ 0 -- 2026-05-12
5032             * LeadingSpace ^ 0

```

```

5033         * ( space ^ 1 * -1 + Main ) ^ 0
5034     * -1
5035     * Lc [[ \@@_end_line: ]]
5036 )

```

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

```

5037 if left_tag then
5038     local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
5039         * Q ( left_tag * other ^ 0 ) -- $
5040         * ( ( ( 1 - P ( right_tag ) ) ^ 0 )
5041             / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
5042         * Q ( right_tag )
5043         * Ct ( Cc "Close" )
5044     MainWithoutTag
5045         = space ^ 1 * -1
5046         + space ^ 0 * EOL
5047         + Space
5048         + Tab
5049         + Escape + EscapeMath
5050         + CommentLaTeX
5051         + Beamer
5052         + DetectedCommands
5053         + sComment
5054         + Delim
5055         + LongString
5056         + PrefixedKeyword
5057         + Keyword * ( -1 + # ( 1 - alphanum ) )
5058         + Punct
5059         + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5060         + Number
5061         + Word
5062     LPEG0[lang] = MainWithoutTag ^ 0
5063     local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
5064                     + Beamer + DetectedCommands + sComment + Tag
5065     MainWithTag
5066         = space ^ 1 * -1
5067         + space ^ 0 * EOL
5068         + Space
5069         + LPEGaux
5070         + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
5071     LPEG1[lang] = MainWithTag ^ 0
5072     LPEG2[lang] =
5073         Ct (
5074             ( space ^ 0 * P "\r" ) ^ -1
5075             * Lc [[ \@@_begin_line: ]]
5076             * Beamer
5077             * LeadingSpace ^ 0
5078             * LPEG1[lang]
5079             * -1
5080             * Lc [[ \@@_end_line: ]]
5081         )
5082 end
5083 end

```

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

```

5084 function piton.write_files_now ( )
5085     for file_name , file_content in pairs ( piton.write_files ) do
5086         local file = io.open ( file_name , "w" )
5087         if file then
5088             file : write ( file_content )

```

```

5089     file : close ( )
5090 else
5091     sprintL3
5092     ( [[ \@@_error_or_warning:nn { FileError } { ]] .. file_name .. "}" )
5093 end
5094 end
5095 end

```

### 3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

5096 function piton.utf16 ( str )
5097     local hex = { "FEFF" } -- BOM UTF-16BE
5098     for _, codepoint in utf8.codes(str) do
5099         table.insert(hex, string.format("%04X", codepoint))
5100     end
5101     return table.concat(hex)
5102 end
5103 </LUA>

```