

The package `piton`*

F. Pantigny
fpantigny@wanadoo.fr

October 11, 2026

Abstract

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

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

1 Introduction

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

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

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

Consider, for example, the following Python code:

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

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

*This document corresponds to the version 4.14c of `piton`, at the date of 2026/10/11.

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

```

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

```

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

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

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

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

```

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

```

2 The L3 part of the implementation

2.1 Declaration of the package

```

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

```

```

11   You~need~at~least~the~version~of~2026-06-01.  \\\
12   If~you~use~Overleaf,~you~need~at~least~"TeX-Live-2026".\\
13   The~package~'piton'~won't~be~loaded.
14 }
15 \IfFormatAtLeastTF
16 { 2026-06-01 }
17 { }
18 { \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.

```

19 \RequirePackage { amstext }

```

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

```

20 \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).

```

21 \RequirePackage { transparent }

```

```

22 \AddToHook { package / babel / after } { \babelprovide { english } }

```

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

```

48 {
49   LuaLaTeX~is~mandatory.\\
50   The~package~'piton'~requires~the~engine~LuaLaTeX.\\
51   \str_if_eq:onT \c_sys_jobname_str { output }
52   { If~you~use~Overleaf,~you~can~switch~to~LuaLaTeX~in~
53     "File~>~Settings~>~Compiler"~and~if~you~use~TeXPage,
54     ~you~should~go~in~"Settings". \\ }
55   \IfClassLoadedT { beamer }
56   {
57     Since~you~use~Beamer,~don't~forget~to~use~piton~in~frames~with~
58     the~key~'fragile'.\\
59   }
60   \IfClassLoadedT { ltx-talk }
61   {
62     Since~you~use~'ltx-talk',~don't~forget~to~use~piton~in~
63     environments~'frame*'.\\
64   }
65 }
66 \sys_if_engine_luatex:F { \@@_fatal:n { LuaLaTeX~mandatory } }

67 \RequirePackage { luacode }

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

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

```

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

```

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

```

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

```

100 \ProcessKeyOptions

```

```

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

114 \RequirePackage { xcolor }

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

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

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

```

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

```

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

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

```

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

```

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

```

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

2.2 Parameters and technical definitions

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

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

```
157 \dim_new:N \l_@@_tmpc_dim
```

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

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

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

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

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

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

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

```
161 \str_set:Nn \l_piton_language_str { python }
```

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

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

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

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

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

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

```
165 \str_new:N \l_@@_join_str
```

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

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

The following parameter corresponds to the key `box`.

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

In order to have a better control over the keys.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The argument of `\PitonInputFile`.

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

The following line can't be deleted.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `\l_@@_left_margin_dim` (for the numbers of lines);
- a small margin when `background-color` is in force²).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

These macros must *not* be protected.

```

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

```

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

```

266 \tl_new:N \g_@@_after_line_tl

```

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

```

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

```

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

```

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

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

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

```

290 \tl_new:N \g_@@_def_vertical_commands_tl

```

The following commands must *not* be protected.

```

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

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

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

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

```

2.4 Treatment of a line of code

```

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

```

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

```

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

```

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

The first implementation was using `\tl_regex_replace_all:nnN`

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

but that programming was certainly slow.

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

```

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

```

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

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

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

```

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

```

```

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

```

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

```

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

```

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

```

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

```

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

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

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

```

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

```

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

```

475         \group_begin:
476         \@@_compute_and_set_color:

```

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

```

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

```

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

```

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

```

Now, the colored panel.

```

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

```

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

```

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

```

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

```

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

```

The group is for the color of the background.

```

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

```

```

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

```

End of \@@_add_bg_and_right_nb_to_line_and_use:

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

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

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

```

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

```

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

```

595   \int_gincr:N \g_@@_line_int

```

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

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

```

596   \par

```

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

```

597   \kern -2.5 pt

```

Now, we control page breaks after the paragraph.

```

598   \@@_add_penalty_for_the_line:
599 }

```

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

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

```

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

```

```

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

```

2.5 PitonOptions

```

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

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

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

```

```

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

```

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

Now, general keys.

```

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

```

```

796     } ,
797     path .value_required:n = true ,
The initial value of the key path is not empty: it's ., that is to say a comma separated list with only
one component which is ., the current directory.
798     path .initial:n = . ,
799     path-write .str_set:N = \l_@@_path_write_str ,
800     path-write .value_required:n = true ,
801     font-command .tl_set:N = \l_@@_font_command_tl ,
802     font-command .initial:n = \ttfamily ,
803     font-command .value_required:n = true ,
804     font-command+ .code:n
805     = { \tl_put_right:Nn \l_@@_font_command_tl { #1 } } ,
806     font-command+ .value_required:n = true ,
807     font-command~+ .meta:n = { font-command+ = #1 } ,
808     font-command~+ .value_required:n = true ,
809     gobble .int_set:N = \l_@@_gobble_int ,
810     gobble .default:n = -1 ,
811     auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
812     auto-gobble .value_forbidden:n = true ,
813     env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
814     env-gobble .value_forbidden:n = true ,
815     tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
816     tabs-auto-gobble .value_forbidden:n = true ,
817
818     splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
819
820     split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,

```

When the key `split-on-empty-lines` is in force, the correspondint token list will be inserted between the chunks of code (the computer listing provided by the end user is split in chunks on the empty lines in the code). That parameter must contain elements to be inserted in *vertical* mode by TeX.

```

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

```

The following line is mandatory.

```

841     line-numbers .default:n = true ,

```

The following counter corresponds to the key `splittable` of `\PitonOptions`. If the value of `\l_@@_splittable_int` is equal to n , then no line break can occur within the first n lines or the last n lines of a listing (or a *chunk* of listings when the key `split-on-empty-lines` is in force).

```

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

```

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

```

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

```

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

```

849     prompt-background-color .tl_set:N      = \l_@@_prompt_bg_color_tl ,
850     prompt-background-color .value_required:n = true ,
851     prompt-background-color .initial:n      = gray!15 ,

```

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

```

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

```

```

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

```

```

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

```

```

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

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

```

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

```

1037 \NewDocumentCommand \@@_fake_PitonOptions { }
1038 { \keys_set:nn { PitonOptions } }

```

2.6 The numbers of the lines

The following counter will be used to count the lines in the code when the user requires the numbers of the lines to be printed (with `line-numbers`) whereas the counter `\g_@@_line_int` previously defined is *not* used for that functionality.

```

1039 \int_new:N \g_@@_visual_line_int
1040 \cs_new_protected:Npn \@@_incr_visual_line:
1041 { \bool_if:NF \l_@@_skip_empty_lines_bool { \int_gincr:N \g_@@_visual_line_int } }

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

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

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

2.7 The main commands and environments for the end user

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

The last argument is provided by curryfication.

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

The two last arguments are provided by curryfication.

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

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

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

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

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

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

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

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

```
1101 }
```

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

```
1102 \keys_set:nn { NewPitonLanguage } { #3 }
```

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

```
1103 \prop_gput:Non \g_@@_languages_prop \l_tmpa_tl { #3 }
```

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

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

```
1105 }
```

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

```
1107 {
```

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

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

```
1110 }
```

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

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

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

```
1113 {
```

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

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

```
1115   {
```

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

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

```
1118   }
```

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

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

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

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

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

```
1122 }
```

```
1123 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4
```

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

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

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

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

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

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

```
1129 {
```

```
1130   \group_begin:
```

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

```
1132   {
```

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

```
1133   \bool_lazy_or:nnT
```

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

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

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

```
1137 }
```

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

```
1138 \automatichyphenmode = 1
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```

1185 \group_end:
1186 }

```

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

```

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

```

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

```

1209 \cs_new_protected:Npn \@@_pre_composition:
1210 {
1211   \dim_compare:nNt \l_@@_width_dim = \c_zero_dim
1212   {
1213     \dim_set_eq:NN \l_@@_width_dim \linewidth

```

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

When the parameters `line-numbers`, `line-numbers/position=left` and `left-margin` are in force (or if `line-numbers`, `line-numbers=right` and `right-margin` are in force), we have to compute the width of the maximal number of lines at the end of the environment to fix the correct value to `left-margin` (or `right-margin`).

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

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

```

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

```

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

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

If there is a background (even a background with the color `none`), we subtract 0.5 em on both sides. However, if there is a left margin or a right margin, we use those margins. If the key `left-margin`

has been used with the special value `auto` (this is meaningful only in conjunction with the key `line-numbers` and a value of `line-numbers/position` equal to `left`), the actual value for the left margin has yet computed (and stored in `left-margin`). Idem for the right margin.

```

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

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

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

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

```

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

```

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

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

```

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

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

```

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

```

1327 \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1328 {
1329
1330 \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1331 #4
1332 \@@_pre_composition:
1333 \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1334 {
1335 \int_gset:Nn \g_@@_visual_line_int
1336 { \l_@@_number_lines_start_int - 1 }
1337 }
1338 \bool_if:NT \g_@@_beamer_bool
1339 { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1340 \bool_if:NT \g_@@_footnote_bool \savenotes
1341 \IfPackageLoadedT { babel }
1342 { \begin { otherlanguage } { english } }
1343 \@@_composition:
1344 \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1345 { \@@_create_paperclip_or_annotation: }
1346 \IfPackageLoadedT { babel } { \end { otherlanguage } }
1347 \bool_if:NT \g_@@_footnote_bool \endsavenotes
1348 #5
1349 }
1350 { \ignorespacesafterend }
1351 }

```

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

```

1352 \cs_new_protected:Npn \@@_create_paperclip_or_annotation:
1353 {
1354 \marginalia [ pos = right ]
1355 {
1356 \vspace* { - 0.8 em }
1357 \hbox:n
1358 {
1359 \vrule~height~0~pt~depth~12~pt~width~0~pt
1360 \bool_if:NT \l_@@_annotation_bool
1361 {
1362 \lua_now:n
1363 {

```

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!

```

1364 pdf.immediateobj
1365 ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1366 }
1367 \pdfextension annot~width~5pt~height~10pt~depth~0pt
1368 {
1369 /Subtype /Text
1370 /Contents~\pdf_object_ref_last:
1371 /Name /Note
1372 /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).

```

1373 /ReplyType /Group

```

Adds the bit 10 which means `LockedContents`.

```

1374 /F~512
1375 /C [0.8~0.8~0.8]
1376 }
1377 \hspace* { 7 mm }
1378 }
1379 \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1380 }

```

```

1381     }
1382 }

1383 \cs_new_protected:Npn \@@_create_paperclip:
1384 {
1385     \str_if_empty:NT \l_@@_paperclip_str
1386     {
1387         \int_gincr:N \g_@@_paperclip_int
1388         \str_set:Nx \l_@@_paperclip_str { listing_\int_use:N \g_@@_paperclip_int .txt }
1389     }

```

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

```

1390     \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1391     \box_move_down:nn
1392     { 10 pt }
1393     {
1394         \hbox:n
1395         {
1396             \pdfextension annot~width~10pt~height~20pt~depth~0pt
1397             {
1398                 /Subtype /FileAttachment
1399                 /Name /Paperclip
1400                 /F-8 % no zoom

```

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

```

1401             /Contents (The~computer~listing)
1402             /FS <<
1403                 /Type /Filespec
1404                 /F (\l_@@_paperclip_str)
1405                 /EF << /F~\pdf_object_ref_last: >>
1406                 /AFRelationship /Supplement
1407             >>
1408         }
1409     }
1410 }
1411 }

```

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

```

1412 \NewDocumentCommand { \NewPitonEnvironment } { }
1413 { \@@_DefinePitonEnvironment:nnnnn { New } }

1414 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1415 { \@@_DefinePitonEnvironment:nnnnn { Declare } }

1416 \NewDocumentCommand { \RenewPitonEnvironment } { }
1417 { \@@_DefinePitonEnvironment:nnnnn { Renew } }

1418 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1419 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

1420 \cs_new_protected:Npn \@@_translate_beamer_env:n
1421 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1422 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1423 \cs_new_protected:Npn \@@_composition:
1424 {
1425     \str_if_empty:NT \l_@@_box_str
1426     {
1427         \mode_if_vertical:F
1428         { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1429     }

1430     \bool_if:NT \l_@@_line_numbers_bool
1431     {
1432         \bool_lazy_and:nnT
1433         { \l_@@_left_margin_auto_bool }

```

```

1434     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { left } }
1435     { @@_compute_margin:N \l_@@_left_margin_dim }
1436     \bool_lazy_and:nnT
1437     { \l_@@_right_margin_auto_bool }
1438     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { right } }
1439     { @@_compute_margin:N \l_@@_right_margin_dim }
1440   }
1441   \lua_now:e
1442   {
1443     piton.join_separation = "\l_@@_join_separation_str"
1444     piton.join = "\l_@@_join_str"
1445     piton.write = "\l_@@_write_str"
1446     piton.path_write = "\l_@@_path_write_str"
1447   }
1448   \noindent
1449   \bool_if:NTF \l_@@_print_bool
1450   {

```

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.

```

1451     \bool_if:NTF \l_@@_split_on_empty_lines_bool
1452     { \par @@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1453     {
1454       @@_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).

```

1455     \bool_if:NTF \l_@@_tcolorbox_bool
1456     {
1457       \str_if_empty:NTF \l_@@_box_str
1458       @@_composition_iii:
1459       @@_composition_iv:
1460     }
1461     {
1462       \str_if_empty:NTF \l_@@_box_str
1463       @@_composition_i:
1464       @@_composition_ii:
1465     }
1466   }
1467 }
1468 { @@_gobble_parse_no_print:o \l_@@_listing_tl }
1469 }

```

`@@_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=...`.

```

1470 \cs_new_protected:Npn @@_composition_i:
1471 {

```

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.

```

1472   \box_clear:N \g_tmpa_box

```

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

```

1473   \box_clear_new:N \g_@@_line_box

```

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

```

1474   \vbox_set:Nn \l_tmpa_box
1475   {
1476     \vbox_unpack_drop:N \g_@@_output_box
1477     \bool_gset_false:N \g_tmpa_bool

```

```

1478     \unskip \unskip
1479     \bool_gset_false:N \g_tmpa_bool
1480     \bool_do_until:n \g_tmpa_bool
1481     {
1482         \unskip \unskip \unskip
1483         \unpenalty \unkern
1484         \box_set_to_last:N \l_@@_line_box
1485         \box_if_empty:N \l_@@_line_box
1486         { \bool_gset_true:N \g_tmpa_bool }
1487         {
1488             \vbox_gset:Nn \g_tmpa_box
1489             {
1490                 \vbox_unpack:N \g_tmpa_box
1491                 \box_use:N \l_@@_line_box
1492             }
1493         }
1494     }
1495 }

```

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

```

1496     \bool_gset_false:N \g_tmpa_bool
1497     \int_zero:N \g_@@_line_int
1498     \bool_do_until:n \g_tmpa_bool
1499     {

```

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`.

```

1500         \vbox_gset:Nn \g_tmpa_box
1501         {
1502             \vbox_unpack_drop:N \g_tmpa_box
1503             \box_gset_to_last:N \g_@@_line_box
1504         }

```

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.

```

1505         \box_if_empty:N \g_@@_line_box
1506         { \bool_gset_true:N \g_tmpa_bool }
1507         {
1508             \box_use:N \g_@@_line_box
1509             \int_gincr:N \g_@@_line_int
1510             \par
1511             \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`.

```

1512         \@@_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.

```

1513         \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int }
1514         { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int } }
1515         \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1516         { \mode_leave_vertical: }
1517     }
1518 }
1519 \skip_vertical:n { 2.5 pt }
1520 }

```

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

```

1521 \cs_new_protected:Npn \@@_composition_ii:
1522 {
1523     \use:e { \begin { minipage } [ \l_@@_box_str ] }
1524     { \l_@@_listing_width_dim }

```

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

```

1525     \vbox_unpack:N \g_@@_output_box

```

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

```

1526   \kern 2.5 pt
1527   \end { minipage }
1528 }

```

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

```

1529 \cs_new_protected:Npn \@@_composition_iii:
1530 {
1531   \use:e
1532   {
1533     \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.

```

1534       [ breakable , text~width = \l_@@_listing_width_dim ]
1535   }
1536   \par
1537   \vbox_unpack:N \g_@@_output_box
1538   \end { tcolorbox }
1539 }

```

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

```

1540 \cs_new_protected:Npn \@@_composition_iv:
1541 {
1542   \use:e
1543   {
1544     \begin { tcolorbox }
1545     [
1546       hbox ,
1547       text~width = \l_@@_listing_width_dim ,
1548       nobeforeafter ,
1549       box~align =
1550         \str_case:Nn \l_@@_box_str
1551         {
1552           t { top }
1553           b { bottom }
1554           c { center }
1555           m { center }
1556         }
1557     ]
1558   }
1559   \box_use:N \g_@@_output_box
1560   \end { tcolorbox }
1561 }

```

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.

```

1562 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1563 {
1564   \int_case:nn
1565   {
1566     \lua_now:e
1567     {
1568       tex.sprint
1569       ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1570     }
1571   }
1572   { 1 { \penalty 100 } 2 \nobreak }
1573 }

```

`\@@_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`.

```

1574 \cs_new_protected:Npn \@@_create_output_box:
1575 {
1576   \@@_compute_code_width:
1577   \vbox_gset:Nn \g_@@_output_box
1578   { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1579   \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1580   \bool_lazy_any:nT
1581   {
1582     { \int_compare_p:nNn \l_@@_bg_colors_int > \c_zero_int }
1583     { \g_@@_rowcolor_inside_bool }
1584     {
1585       \l_@@_line_numbers_bool
1586       &&
1587       \str_if_eq_p:ee \l_@@_line_numbers_position_str { right }
1588     }
1589   }
1590   \@@_add_bg_and_right_nb_to_output_box:
1591 }

```

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:`.

```

1592 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_output_box:
1593 {
1594   \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`.

```

1595   \vbox_set:Nn \l_tmpa_box
1596   {
1597     \vbox_unpack_drop:N \g_@@_output_box

```

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

```

1598     \bool_gset_false:N \g_tmpa_bool
1599     \unskip \unskip

```

We begin the loop.

```

1600     \bool_do_until:nn \g_tmpa_bool
1601     {
1602       \unskip \unskip \unskip
1603       \int_set_eq:NN \l_tmpa_int \lastpenalty
1604       \unpenalty \unkern

```

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

```

1605       \box_set_to_last:N \l_@@_line_box
1606       \box_if_empty:NTF \l_@@_line_box
1607       { \bool_gset_true:N \g_tmpa_bool }
1608       {

```

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

```

1609       \vbox_gset:Nn \g_@@_output_box
1610       {

```

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`.

```

1611       \@@_add_bg_and_right_nb_to_line_and_use:
1612       \kern -2.5 pt
1613       \penalty \l_tmpa_int
1614       \vbox_unpack:N \g_@@_output_box
1615     }

```

```

1616         }
1617         \int_gdecr:N \g_@@_line_int
1618     }
1619 }
1620 }

```

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

```

1621 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1622 {
1623     \lua_now:e
1624     {
1625         piton.GobbleParseNoPrint
1626         (
1627             '\l_piton_language_str' ,
1628             \int_use:N \l_@@_gobble_int ,
1629             token.scan_argument ( )
1630         )
1631     }
1632 }
1633 \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.

```

1634 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1635 {
1636     \lua_now:e
1637     {
1638         piton.RetrieveGobbleParse
1639         (
1640             '\l_piton_language_str' ,
1641             \int_use:N \l_@@_gobble_int ,
1642             \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1643             { \int_eval:n { - \l_@@_splittable_int } }
1644             { \int_use:N \l_@@_splittable_int } ,
1645             token.scan_argument ( )
1646         )
1647     }
1648 }
1649 \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.

```

1650 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1651 {
1652     \lua_now:e
1653     {
1654         piton.RetrieveGobbleSplitParse
1655         (
1656             '\l_piton_language_str' ,
1657             \int_use:N \l_@@_gobble_int ,
1658             \int_use:N \l_@@_splittable_int ,
1659             token.scan_argument ( )
1660         )
1661     }
1662 }
1663 \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`.

```

1664 \bool_if:NTF \g_@@_beamer_bool

```

```

1665 {
1666   \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1667   {
1668     \keys_set:nn { PitonOptions } { #2 }
1669     \begin { actionenv } < #1 >
1670   }
1671   { \end { actionenv } }
1672 }
1673 {
1674   \NewPitonEnvironment { Piton } { 0 { } }
1675   { \keys_set:nn { PitonOptions } { #1 } }
1676   { }
1677 }

1678 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1679 {
1680   \mode_if_vertical:F { \par }
1681   \group_begin:
1682   \seq_concat:NNN
1683     \l_file_search_path_seq
1684     \l_@@_path_seq
1685     \l_file_search_path_seq
1686   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1687   {
1688     \@@_input_file:nn { #1 } { #2 }
1689     #4
1690   }
1691   { #5 }
1692   \group_end:
1693 }

1694 \cs_new_protected:Npn \@@_unknown_file:n #1
1695 { \msg_error:nnn { piton } { Unknown~file } { #1 } }
1696 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1697 {
1698   \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1699   {

```

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

```

1700   \iow_log:n { No~file~#3 }
1701   \@@_unknown_file:n { #3 }
1702 }
1703 }
1704 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1705 {
1706   \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1707   {

```

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

```

1708   \iow_log:n { No~file~#3 }
1709   \@@_unknown_file:n { #3 }
1710 }
1711 }
1712 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1713 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```

1714 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1715 {

```

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 >).

```

1716   \tl_if_novalue:nF { #1 }
1717   {
1718     \bool_if:NTF \g_@@_beamer_bool

```

```

1719     { \begin { uncoverenv } < #1 > }
1720     { \@@_error_or_warning:n { overlay~without~beamer } }
1721 }
1722 \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.

```

1723 \iow_log:e { (\l_@@_file_name_str) }
1724 \int_zero_new:N \l_@@_first_line_int
1725 \int_zero_new:N \l_@@_last_line_int
1726 \int_set_eq:NN \l_@@_last_line_int \c_max_int
1727 \bool_set_true:N \l_@@_in_PitonInputFile_bool
1728 \keys_set:nn { PitonOptions } { #2 }
1729 \bool_if:NT \l_@@_line_numbers_absolute_bool
1730 { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1731 \bool_if:nTF
1732 {
1733   (
1734     \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1735     || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1736   )
1737   && ! \str_if_empty_p:N \l_@@_begin_range_str
1738 }
1739 {
1740   \@@_error_or_warning:n { bad~range~specification }
1741   \int_zero:N \l_@@_first_line_int
1742   \int_set_eq:NN \l_@@_last_line_int \c_max_int
1743 }
1744 {
1745   \str_if_empty:NF \l_@@_begin_range_str
1746   {
1747     \@@_compute_range:
1748     \bool_lazy_or:nnT
1749       \l_@@_marker_include_lines_bool
1750       { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1751       {
1752         \int_decr:N \l_@@_first_line_int
1753         \int_incr:N \l_@@_last_line_int
1754       }
1755   }
1756 }
1757 \@@_pre_composition:
1758 \bool_if:NT \l_@@_line_numbers_absolute_bool
1759 { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1760 \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1761 {
1762   \int_gset:Nn \g_@@_visual_line_int
1763   { \l_@@_number_lines_start_int - 1 }
1764 }

```

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

```

1765 \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1766 { \int_gzero:N \g_@@_visual_line_int }
1767 \lua_now:e
1768 {

```

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

```

1769 piton.ReadFile(
1770   '\l_@@_file_name_str' ,
1771   \int_use:N \l_@@_first_line_int ,
1772   \int_use:N \l_@@_last_line_int )
1773 }
1774 \@@_composition:

```

```
1775 \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.

```
1776 \tl_if_novalue:nF { #1 }
1777 { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1778 }
```

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

```
1779 \cs_new_protected:Npn \@@_compute_range:
1780 {
```

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

```
1781 \str_set:Nx \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1782 \str_set:Nx \l_tmpb_str { \@@_marker_end:n { \l_@@_end_range_str } }
```

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

```
1783 \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1784 \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1785 \lua_now:e
1786 {
1787     piton.ComputeRange
1788     ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1789 }
1790 }
```

2.8 The styles

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

```
1791 \NewDocumentCommand { \PitonStyle } { m }
1792 {
1793     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1794     { \use:c { pitonStyle _ #1 } }
1795 }
```

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”.

```
1796 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1797 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1798 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1799 {
1800     \str_clear_new:N \l_@@_SetPitonStyle_option_str
1801     \str_set:Nx \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1802     \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1803     { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1804     \keys_set:nn { piton / Styles } { #2 }
1805 }

1806 \cs_new_protected:Npn \@@_math_scantokens:n #1
1807 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

1808 \clist_new:N \g_@@_styles_clist
1809 \clist_gset:Nn \g_@@_styles_clist
1810 {
1811     Comment ,
1812     Comment.Internal ,
1813     Comment.LaTeX ,
1814     Discard ,
```

```

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

```

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

```

1860 TypeExpression ,

```

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

```

1861 Directive
1862 }

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

```

```

1875         }
1876         { ##1 }
1877     }
1878 }
1879
1880 \keys_define:nn { piton / Styles }
1881 {
1882     String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1883     String      .value_required:n = true ,
1884     Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1885     Comment.Math .value_required:n = true ,
1886     unknown     .code:n = \@@_unknown_style:
1887 }

```

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.

```

1888 \cs_new_protected:Npn \@@_unknown_style:
1889 {
1890     \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1891     {
1892         \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1893         \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`.

```

1894         \bool_lazy_and:nnTF
1895         { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1896         {
1897             \str_if_eq_p:Vn \l_tmpa_str { Module }
1898             ||
1899             \str_if_eq_p:Vn \l_tmpa_str { Type }
1900         }

```

Now, we will create a new style.

```

1901         { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1902         { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1903     }
1904     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1905 }

```

```

1906 \SetPitonStyle[OCaml]
1907 {
1908     TypeExpression =
1909     {
1910         \SetPitonStyle [ OCaml ]
1911         {
1912             Identifier = \PitonStyle { Name.Type } ,
1913             Name.Builtin = \PitonStyle { Name.Type}
1914         }
1915         \@@_piton:n
1916     }
1917 }

```

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`.

```

1918 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that `clist`.

```

1919 \clist_gsort:Nn \g_@@_styles_clist
1920 {
1921     \str_compare:nNnTF { #1 } < { #2 }
1922     \sort_return_same:
1923     \sort_return_swapped:
1924 }

```

```

1925 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1926
1927 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1928
1929 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1930 {
1931   \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`.

```

1932   \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl \space
1933   \seq_clear:N \l_tmpa_seq
1934   \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1935   \seq_use:Nn \l_tmpa_seq { \- }
1936 }

```

```

1937 \cs_new_protected:Npn \@@_comment:n #1
1938 { \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”.

```

1939 \cs_new_protected:Npn \PitonSpaceSubstitute #1 % noqa
1940 {
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 \cs_new_protected:Npn \@@_string_long:n #1
1952 {
1953   \PitonStyle { String.Long }
1954   {
1955     \bool_if:NTF \l_@@_break_strings_anywhere_bool
1956     { \@@_actually_break_anywhere:n { #1 } }
1957     {

```

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.

```

1958       \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1959       {
1960         \tl_set:Nn \l_tmpa_tl { #1 }
1961         \tl_replace_all:Nvn \l_tmpa_tl
1962           \c_catcode_other_space_tl
1963           \@@_breakable_space:
1964         \l_tmpa_tl
1965       }
1966       { #1 }
1967     }
1968   }
1969 }

```

```

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

```

2.9 The initial styles

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

```

2003 \SetPitonStyle
2004 {
2005   Comment           = \color [ HTML ] { 0099FF } \itshape ,
2006   Comment.Internal  = \@@_comment:n ,
2007   Exception         = \color [ HTML ] { CC0000 } ,
2008   Keyword           = \color [ HTML ] { 006699 } \bfseries ,
2009   Keyword.Governing = \color [ HTML ] { 006699 } \bfseries ,
2010   Keyword.Constant  = \color [ HTML ] { 006699 } \bfseries ,
2011   Name.Builtin      = \color [ HTML ] { 336666 } ,
2012   Name.Decorator     = \color [ HTML ] { 9999FF } ,
2013   Name.Class        = \color [ HTML ] { 00AA88 } \bfseries ,
2014   Name.Function      = \color [ HTML ] { CC00FF } ,
2015   Name.Namespace    = \color [ HTML ] { 00CCFF } ,
2016   Name.Constructor   = \color [ HTML ] { 006000 } \bfseries ,
2017   Name.Field        = \color [ HTML ] { AA6600 } ,
2018   Name.Module       = \color [ HTML ] { 0060A0 } \bfseries ,
2019   Name.Table        = \color [ HTML ] { 309030 } ,
2020   Number            = \color [ HTML ] { FF6600 } ,
2021   Number.Internal   = \@@_number:n ,
2022   Operator          = \color [ HTML ] { 555555 } ,
2023   Operator.Word     = \bfseries ,
2024   String            = \color [ HTML ] { CC3300 } ,
2025   String.Long.Internal = \@@_string_long:n ,
2026   String.Short.Internal = \@@_string_short:n ,

```

```

2027 String.Doc.Internal = \@@_string_doc:n ,
2028 String.Doc          = \color [ HTML ] { CC3300 } \itshape ,
2029 String.Interpol     = \color [ HTML ] { AA0000 } ,
2030 Comment.LaTeX       = \normalfont \color [ rgb ] { .468, .532, .6 } ,
2031 Name.Type           = \color [ HTML ] { 336666 } ,
2032 InitialValues       = \@@_piton:n ,
2033 Interpol.Inside     = { \l_@@_font_command_tl \@@_piton:n } ,
2034 TypeParameter       = \color [ HTML ] { 336666 } \itshape ,
2035 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).

```

2036 Identifier.Internal = \@@_identifier:n ,
2037 Identifier          = ,
2038 Directive           = \color [ HTML ] { AA6600 } ,
2039 Tag                 = \colorbox { gray!10 } ,
2040 UserFunction        = \PitonStyle { Identifier } ,
2041 Prompt              = ,
2042 Discard             = \use_none:n
2043 }

```

2.10 Styles specific to the language expl

```

2044 \clist_new:N \g_@@_expl_styles_clist
2045 \clist_gset:Nn \g_@@_expl_styles_clist
2046 {
2047   Scope.l ,
2048   Scope.g ,
2049   Scope.c
2050 }
2051 \clist_map_inline:Nn \g_@@_expl_styles_clist
2052 {
2053   \keys_define:nm { piton / Styles }
2054   {
2055     #1 .value_required:n = true ,
2056     #1 .code:n =
2057       \tl_set:cn
2058       {
2059         pitonStyle _
2060         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
2061         { \l_@@_SetPitonStyle_option_str _ }
2062         #1
2063       }
2064     { ##1 }
2065   }
2066 }
2067 \SetPitonStyle [ expl ]
2068 {
2069   Scope.l      = ,
2070   Scope.g      = \bfseries ,
2071   Scope.c      = \slshape ,
2072   Type.bool    = \color [ HTML ] { AA6600 } ,
2073   Type.box     = \color [ HTML ] { 267910 } ,
2074   Type.clist   = \color [ HTML ] { 309030 } ,
2075   Type.fp      = \color [ HTML ] { FF3300 } ,
2076   Type.int     = \color [ HTML ] { FF6600 } ,
2077   Type.seq     = \color [ HTML ] { 309030 } ,
2078   Type.skip    = \color [ HTML ] { 0CC060 } ,
2079   Type.str     = \color [ HTML ] { CC3300 } ,
2080   Type.tl      = \color [ HTML ] { AA2200 } ,

```

```

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

```

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)].

```

2096 \hook_gput_code:nnn { begindocument } { . }
2097 {
2098   \bool_if:NT \g_@@_math_comments_bool
2099     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
2100 }

```

2.11 Highlighting some identifiers

```

2101 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
2102 {
2103   \clist_set:Nn \l_tmpa_clist { #2 }
2104   \tl_if_novalue:nTF { #1 }
2105     {
2106       \clist_map_inline:Nn \l_tmpa_clist
2107         { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
2108     }
2109     {
2110       \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
2111       \str_if_eq:onT \l_tmpa_str { current-language }
2112         { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
2113       \clist_map_inline:Nn \l_tmpa_clist
2114         { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
2115     }
2116 }
2117 \cs_new_protected:Npn \@@_identifier:n #1
2118 {
2119   \str_set:Nn \l_tmpa_str { #1 }
2120   \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2121     {
2122       \cs_if_exist_use:cF { PitonIdentifier _ \l_tmpa_str }
2123         { \PitonStyle { Identifier } }
2124     }
2125   { #1 }
2126 }

```

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`).

```

2127 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
2128 {

```

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

```
2129 { \PitonStyle { Name.Function } { #1 } }
2130 \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}`.

```
2131 \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2132 { \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`.**

```
2133 \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
2134 { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
2135 \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.

```
2136 \seq_if_in:NoF \g_@@_languages_seq { \l_piton_language_str }
2137 { \seq_gput_left:No \g_@@_languages_seq { \l_piton_language_str } }
2138 }
```

```
2139 \NewDocumentCommand \PitonClearUserFunctions { ! o }
2140 {
2141   \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.

```
2142 { \@@_clear_all_functions: }
2143 { \@@_clear_list_functions:n { #1 } }
2144 }
```

```
2145 \cs_new_protected:Npn \@@_clear_list_functions:n #1
2146 {
2147   \clist_set:Nn \l_tmpa_clist { #1 }
2148   \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
2149   \clist_map_inline:nn { #1 }
2150   { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
2151 }
```

```
2152 \cs_new_protected:Npn \@@_clear_functions_i:n #1
2153 { \@@_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).

```
2154 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2155 {
2156   \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2157   {
2158     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2159     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }
2160     \seq_gclear:c { g_@@_functions _ #1 _ seq }
2161   }
2162 }
2163 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }
```

```
2164 \cs_new_protected:Npn \@@_clear_functions:n #1
2165 {
2166   \@@_clear_functions_i:n { #1 }
2167   \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2168 }
```

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

```

2169 \cs_new_protected:Npn \@@_clear_all_functions:
2170 {
2171   \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2172   \seq_gclear:N \g_@@_languages_seq
2173 }

```

```

2174 \AtEndDocument
2175 {

```

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

```

2176   \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.

```

2177   \IfPDFManagementActiveTF
2178     { \@@_join_files: }
2179     { \@@_join_files_legacy: }
2180 }

```

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

```

2181 \cs_new_protected:Npn \@@_join_files:
2182 {
2183   \seq_map_inline:Nn \g_@@_join_seq
2184   {
2185     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2186     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2187     \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2188     {
2189       <<
2190         /Type /Filespec
2191         /UF <\l_tmpa_str>
2192         /EF << /F~\pdf_object_ref_last: >>
2193         /Desc (Computer~listing)
2194         /AFRelationship /Supplement
2195       >>
2196     }
2197   }
2198 }

```

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

```

2199 \cs_new_protected:Npn \@@_join_files_legacy:
2200 {
2201   \seq_map_inline:Nn \g_@@_join_seq
2202   {
2203     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2204     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2205     \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`.

```

2206     {
2207       /Subtype /FileAttachment
2208       /F~2
2209       /Name /Paperclip
2210       /Contents (Computer~listing)
2211       /FS <<
2212         /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).

```
2213 /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`).

```
2214 /EF << /F~\pdf_object_ref_last: >>
2215 /AFRelationship /Supplement
2216 >>
2217 }
2218 }
2219 }
```

2.12 Spaces of indentation

```
2220 \cs_new_protected:Npn \@@_define_leading_space_normal:
2221 {
2222   \cs_set_protected:Npn \@@_leading_space:
2223   {
2224     \int_gincr:N \g_@@_indentation_int
```

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

```
2225   \hbox:n { ~ }
2226 }
2227 }
2228 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2229 {
2230   \cs_set_protected:Npn \@@_leading_space:
2231   {
2232     \int_gincr:N \g_@@_indentation_int
2233     \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2234     {
2235       \color { white }
2236       \transparent { 0 }
2237       . % previously : □ U+2423
2238     }
2239     \pdfextension literal { EMC }
2240   }
2241 }
2242 \@@_define_leading_space_Foxit:
```

2.13 Security

```
2243 \AddToHook { env / piton / before }
2244 { \@@_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.

```
2245 \cs_new_protected:Npn \@@_unknown_key:nn #1 #2
2246 {
2247   \str_set_eq:NN \l_tmpa_str \l_keys_key_str
2248   \str_replace_all:Nnn \l_tmpa_str { ~ } { - }
2249   \str_set:Nc \l_tmpa_str { \str_lowercase:f { \l_tmpa_str } }
2250   \bool_set_false:N \l_tmpa_bool
2251   \clist_map_inline:nn { #1 }
2252   {
2253     \keys_if_exist:neT { ##1 } { \l_tmpa_str }
2254     {
```

```

2255         \@@_error:n { key-with-normal-form-exists }
2256         \bool_set_true:N \l_tmpa_bool
2257         \clist_map_break:
2258     }
2259 }
2260 \bool_if:NF \l_tmpa_bool { \@@_error:n { #2 } }
2261 }
2262 \@@_msg_new:nn { key-with-normal-form-exists }
2263 {
2264     The~key~'\l_keys_key_str'~does~not~exists.\\
2265     It~will~be~ignored.~Maybe~you~want~to~use~the~key~'\l_tmpa_str'.
2266 }
2267 \@@_msg_new:nn { No-environment-piton }
2268 {
2269     There~is~no~environment~piton!\\
2270     There~is~an~environment~{Piton}~and~a~command~
2271     \token_to_str:N \piton\ but~there~is~no~environment~
2272     {piton}.
2273 }
2274 \@@_msg_new:nn { rounded-corners-without-Tikz }
2275 {
2276     TikZ~not~used \\
2277     You~can't~use~the~key~'rounded-corners'~because~
2278     you~have~not~loaded~the~package~TikZ.~
2279     If~you~go~on,~your~key~will~be~ignored.~
2280     You~won't~have~similar~error~till~the~end~of~the~document.
2281 }
2282 \@@_msg_new:nn { tcolorbox-not-loaded }
2283 {
2284     tcolorbox~not~loaded \\
2285     You~can't~use~the~key~'tcolorbox'~because~
2286     you~have~not~loaded~the~package~tcolorbox.~
2287     Use~\token_to_str:N \usepackage[breakable]{tcolorbox}.~
2288     If~you~go~on,~that~key~will~be~ignored.
2289 }
2290 \@@_msg_new:nn { library-breakable-not-loaded }
2291 {
2292     breakable~not~loaded \\
2293     You~can't~use~the~key~'tcolorbox'~because~
2294     you~have~not~loaded~the~library~'breakable'~of~tcolorbox'.~
2295     You~should~load~'piton'~with~the~key~'breakable'~or~
2296     use~\token_to_str:N \tcbuselibrary{breakable}~
2297     in~the~preamble~of~your~document.~
2298     If~you~go~on,~that~key~will~be~ignored.
2299 }
2300 \@@_msg_new:nn { Language-not-defined }
2301 {
2302     Language~not~defined \\
2303     The~language~'\l_tmpa_tl'~has~not~been~defined~previously.~
2304     If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2305     will~be~ignored.
2306 }
2307 \@@_msg_new:nn { bad-version-of-piton.lua }
2308 {
2309     Bad~number~version~of~'piton.lua'\\
2310     The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2311     version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2312     address~that~issue.
2313 }
2314 \@@_msg_new:nn { Unknown-key-NewPitonLanguage }

```

```

2315 {
2316     Unknown~key~for~\token_to_str:N \NewPitonLanguage.\\
2317     The~key~'\l_keys_key_str'~is~unknown.~
2318     This~key~will~be~ignored.
2319 }
2320 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
2321 {
2322     The~style~'\l_keys_key_str'~is~unknown.\\
2323     This~setting~will~be~ignored.~
2324     The~available~styles~are~(in~alphabetic~order):~
2325     \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2326 }
2327 \@@_msg_new:nn { Invalid~key }
2328 {
2329     Wrong~use~of~key.\\
2330     You~can't~use~the~key~'\l_keys_key_str'~here.~
2331     Your~key~will~be~ignored.
2332 }
2333 \@@_msg_new:nn { Unknown~key~for~line~numbers }
2334 {
2335     Unknown~key. \\
2336     The~key~'line~numbers / \l_keys_key_str'~is~unknown.~
2337     The~available~keys~of~the~family~'line~numbers'~are~(in~
2338     alphabetic~order):~
2339     absolute,~false,~format,~label~empty~lines,~lmonoid10~drawn,~
2340     ~position,~resume,~skip~empty~lines,~sep,~start~and~true.~
2341     Your~key~will~be~ignored.
2342 }
2343 \@@_msg_new:nn { Unknown~key~for~marker }
2344 {
2345     Unknown~key. \\
2346     The~key~'marker / \l_keys_key_str'~is~unknown.~
2347     The~available~keys~of~the~family~'marker'~are~(in~
2348     alphabetic~order):~ beginning,~end~and~include~lines.~
2349     Your~key~will~be~ignored.
2350 }
2351 \@@_msg_new:nn { bad~range~specification }
2352 {
2353     Incompatible~keys.\\
2354     You~can't~specify~the~range~of~lines~to~include~by~using~both~
2355     markers~and~explicit~number~of~lines.~
2356     Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2357 }
2358 \cs_new_nopar:Nn \@@_thepage:
2359 {
2360     \thepage
2361     \cs_if_exist:NT \insertframenumbers
2362     {
2363         ~(frame~\insertframenumbers
2364         \cs_if_exist:NT \beamer@slidenumbers { ,~slide~\insertslidenumbers }
2365         )
2366     }
2367 }

```

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`.

```

2368 \@@_msg_new:nn { SyntaxError }
2369 {
2370     Syntax~Error~on~page~\@@_thepage:.\\
2371     Your~code~of~the~language~'\l_piton_language_str'~is~not~

```

```

2372 syntactically~correct.~
2373 It~won't~be~printed~in~the~PDF~file.
2374 }

2375 \@@_msg_new:nn { FileError }
2376 {
2377   File~Error.\\
2378   It's~not~possible~to~write~on~the~file~'#1' \\
2379   \sys_if_shell_unrestricted:F
2380   { (try~to~compile~with~'lua~tex~--shell~escape').\\ }
2381   If~you~go~on,~nothing~will~be~written~on~that~file.
2382 }

2383 \@@_msg_new:nn { InexistentDirectory }
2384 {
2385   Inexistent~directory.\\
2386   The~directory~'\l_@@_path_write_str'~
2387   given~in~the~key~'path~write'~does~not~exist.~
2388   Nothing~will~be~written~on~'\l_@@_write_str'.
2389 }

2390 \@@_msg_new:nn { begin~marker~not~found }
2391 {
2392   Marker~not~found.\\
2393   The~range~'\l_@@_begin_range_str'~provided~to~the~
2394   command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
2395   The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
2396 }

2397 \@@_msg_new:nn { end~marker~not~found }
2398 {
2399   Marker~not~found.\\
2400   The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
2401   provided~to~the~command~\token_to_str:N \PitonInputFile\
2402   has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
2403   be~inserted~till~the~end.
2404 }

2405 \@@_msg_new:nn { Unknown~file }
2406 {
2407   Unknown~file. \\
2408   The~file~'#1'~is~unknown.~
2409   Your~command~\token_to_str:N \PitonInputFile\ will~be~ignored.
2410 }

2411 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2412 {
2413   \bool_if:NF \g_@@_beamer_bool
2414   { \@@_error_or_warning:n { Without~beamer } }
2415 }

2416 \@@_msg_new:nn { Without~beamer }
2417 {
2418   Key~'\l_keys_key_str'~without~Beamer.\\
2419   You~should~not~use~the~key~'\l_keys_key_str'~since~you~
2420   are~not~in~Beamer.~However,~you~can~go~on.
2421 }

2422 \@@_msg_new:nn { rowcolor~in~detected~commands }
2423 {
2424   'rowcolor'~forbidden~in~'detected~commands'.\\
2425   You~should~put~'rowcolor'~in~'raw~detected~commands'.~
2426   That~key~will~be~ignored.
2427 }

2428 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2429 {
2430   Unknown~key. \\
2431   The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~

```

```

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

2492 \@@_msg_new:nn { label-with-lines-numbers }
2493 {

```

```

2494   You~can't~use~the~command~\token_to_str:N \label\
2495   or~\token_to_str:N \zlabel\
2496   because~the~key~'line-numbers'~
2497   is~not~active.~If~you~go~on,~that~command~will~ignored.
2498 }

2499 \@@_msg_new:nn { overlay~without~beamer }
2500 {
2501   You~can't~use~an~argument~<...>~for~your~command~
2502   \token_to_str:N \PitonInputFile\ because~you~are~not~
2503   in~Beamer.~If~you~go~on,~that~argument~will~be~ignored.
2504 }

2505 \@@_msg_new:nn { label~as~zlabel~needs~zref~package }
2506 {
2507   The~key~'label-as-zlabel'~requires~the~package~'zref'.~
2508   Please~load~the~package~'zref'~before~setting~the~key.
2509 }
2510 \hook_gput_code:nnn { begindocument } { . }
2511 {
2512   \bool_if:NT \g_@@_label_as_zlabel_bool
2513   {
2514     \IfPackageLoadedF { zref-base }
2515     { \@@_fatal:n { label~as~zlabel~needs~zref~package } }
2516   }
2517 }

```

2.15 The glyphs of the 10 arabic numbers

```

2518 \cs_new_protected:Npn \@@_draw_number:n #1
2519 { \tl_map_inline:nn { #1 } { \use:c { c_@@_glyph_ ##1 _tl } } }
2520 \cs_generate_variant:Nn \@@_draw_number:n { e }

2521 \cs_set_eq:NN \tlconstcn \tl_const:cn

2522 \ExplSyntaxOff
2523 \tlconstcn{c_@@_glyph_0_tl}
2524 {%
2525   \hbox to 5pt
2526   {%
2527     \hfill
2528     \pdfextension literal
2529     {
2530       4.24 3.05 m
2531       4.24 4.91 3.22 6.22 2.12 6.22 c
2532       1.00 6.22 0 4.88 0 3.06 c
2533       0 1.20 1.02 -0.11 2.12 -0.11 c
2534       3.24 -0.11 4.24 1.23 4.24 3.05 c
2535       3.55 3.16 m
2536       3.55 1.68 2.90 0.50 2.12 0.50 c
2537       1.34 0.50 0.69 1.68 0.69 3.16 c
2538       0.69 4.62 1.38 5.61 2.12 5.61 c
2539       2.85 5.61 3.55 4.63 3.55 3.16 c
2540       h f
2541     }%
2542     \hfill
2543   }%
2544 }

2545 \tlconstcn{c_@@_glyph_1_tl}
2546 {%
2547   \hbox to 5pt

```

```

2548     {%
2549         \hfill \kern 1 pt
2550         \pdfextension literal
2551         {
2552             3.37 0.3 m
2553             3.37 0.61 3.13 0.61 2.97 0.61 c
2554             2.06 0.61 l
2555             2.06 5.81 l
2556             2.06 5.97 2.06 6.22 1.76 6.22 c
2557             1.57 6.22 1.51 6.1 1.46 5.98 c
2558             1.08 5.13 0.56 5.02 0.37 5.0 c
2559             0.21 4.99 0.0 4.97 0.0 4.69 c
2560             0.0 4.44 0.18 4.39 0.33 4.39 c
2561             0.52 4.39 0.93 4.45 1.37 4.83 c
2562             1.37 0.61 l
2563             0.46 0.61 l
2564             0.3 0.61 0.06 0.61 0.06 0.3 c
2565             0.06 0.0 0.31 0.0 0.46 0.0 c
2566             2.97 0.0 l
2567             3.12 0.0 3.37 0.0 3.37 0.3 c
2568             h f
2569         }%
2570     \hfill
2571 }%
2572 }

2573 \tlconstcn{c_@@_glyph_2_tl}
2574 {%
2575     \hbox to 5pt
2576     {%
2577         \hfill
2578         \pdfextension literal
2579         {
2580             4.2 0.41 m
2581             4.2 0.67 l
2582             4.2 0.86 4.2 1.08 3.86 1.08 c
2583             3.51 1.08 3.51 0.89 3.51 0.61 c
2584             1.13 0.61 l
2585             1.72 1.12 2.68 1.87 3.11 2.27 c
2586             3.74 2.83 4.2 3.47 4.2 4.27 c
2587             4.2 5.47 3.19 6.22 1.97 6.22 c
2588             0.79 6.22 0.0 5.4 0.0 4.55 c
2589             0.0 4.18 0.28 4.07 0.45 4.07 c
2590             0.66 4.07 0.89 4.24 0.89 4.52 c
2591             0.89 4.64 0.84 4.77 0.75 4.84 c
2592             0.9 5.3 1.37 5.61 1.92 5.61 c
2593             2.74 5.61 3.51 5.15 3.51 4.27 c
2594             3.51 3.57 3.02 2.99 2.36 2.44 c
2595             0.15 0.58 l
2596             0.06 0.5 0.0 0.45 0.0 0.31 c
2597             0.0 0.0 0.25 0.0 0.41 0.0 c
2598             3.8 0.0 l
2599             4.13 0.0 4.2 0.09 4.2 0.41 c
2600             h f
2601         }%
2602     \hfill
2603 }%
2604 }

2605 \tlconstcn{c_@@_glyph_3_tl}
2606 {%
2607     \hbox to 5pt
2608     {%
2609         \hfill
2610         \pdfextension literal

```

```

2611 {
2612     4.36 1.74 m
2613     4.36 2.45 3.89 3.04 3.23 3.34 c
2614     3.79 3.7 4.07 4.27 4.07 4.81 c
2615     4.07 5.55 3.34 6.22 2.19 6.22 c
2616     0.99 6.22 0.29 5.74 0.29 5.05 c
2617     0.29 4.72 0.54 4.58 0.74 4.58 c
2618     0.95 4.58 1.18 4.75 1.18 5.03 c
2619     1.18 5.17 1.12 5.27 1.09 5.3 c
2620     1.4 5.61 2.11 5.61 2.2 5.61 c
2621     2.88 5.61 3.38 5.25 3.38 4.8 c
2622     3.38 4.5 3.23 4.15 2.96 3.93 c
2623     2.64 3.67 2.39 3.65 2.03 3.63 c
2624     1.46 3.59 1.31 3.59 1.31 3.3 c
2625     1.31 2.99 1.55 2.99 1.71 2.99 c
2626     2.17 2.99 l
2627     3.16 2.99 3.67 2.32 3.67 1.74 c
2628     3.67 1.13 3.11 0.5 2.2 0.5 c
2629     1.8 0.5 1.03 0.61 0.77 1.08 c
2630     0.82 1.13 0.89 1.19 0.89 1.39 c
2631     0.89 1.63 0.7 1.83 0.45 1.83 c
2632     0.22 1.83 0.0 1.68 0.0 1.36 c
2633     0.0 0.47 0.97 -0.11 2.2 -0.11 c
2634     3.51 -0.11 4.36 0.81 4.36 1.74 c
2635     h f
2636 }%
2637 \hfill
2638 }%
2639 }
2640 \tlconstcn{c_@@_glyph_4_tl}
2641 {%
2642     \hbox to 5pt
2643     {%
2644         \hfill
2645         \pdfextension literal
2646         {
2647             4.66 1.99 m
2648             4.66 2.3 4.42 2.3 4.26 2.3 c
2649             3.48 2.3 l
2650             3.48 5.82 l
2651             3.48 6.15 3.41 6.23 3.07 6.23 c
2652             2.79 6.23 l
2653             2.55 6.23 2.5 6.22 2.37 6.02 c
2654             0.09 2.44 l
2655             0.0 2.31 0.0 2.29 0.0 2.09 c
2656             0.0 1.76 0.09 1.69 0.4 1.69 c
2657             2.92 1.69 l
2658             2.92 0.61 l
2659             2.3 0.61 l
2660             2.14 0.61 1.9 0.61 1.9 0.3 c
2661             1.9 0.0 2.15 0.0 2.3 0.0 c
2662             4.1 0.0 l
2663             4.25 0.0 4.5 0.0 4.5 0.3 c
2664             4.5 0.61 4.26 0.61 4.1 0.61 c
2665             3.48 0.61 l
2666             3.48 1.69 l
2667             4.26 1.69 l
2668             4.41 1.69 4.66 1.69 4.66 1.99 c
2669             2.92 2.3 m
2670             0.7 2.3 l
2671             2.92 5.79 l
2672             h f
2673         }%

```

```

2674     \hfill
2675 }%
2676 }

2677 \tlconstcn{c_@@_glyph_5_tl}
2678 {%
2679     \hbox to 5pt
2680     {%
2681         \hfill
2682         \pdfextension literal
2683         {
2684             4.2 1.9 m
2685             4.2 2.91 3.43 3.89 2.25 3.89 c
2686             1.9 3.89 1.46 3.82 1.05 3.6 c
2687             1.05 5.5 l
2688             3.45 5.5 l
2689             3.6 5.5 3.85 5.5 3.85 5.8 c
2690             3.85 6.11 3.61 6.11 3.45 6.11 c
2691             0.76 6.11 l
2692             0.43 6.11 0.36 6.02 0.36 5.7 c
2693             0.36 3.04 l
2694             0.36 2.86 0.36 2.63 0.68 2.63 c
2695             0.86 2.63 0.9 2.68 0.98 2.78 c
2696             1.25 3.1 1.66 3.28 2.24 3.28 c
2697             3.06 3.28 3.51 2.55 3.51 1.9 c
2698             3.51 1.1 2.8 0.5 1.96 0.5 c
2699             1.68 0.5 1.04 0.58 0.76 1.13 c
2700             0.81 1.18 0.89 1.25 0.89 1.45 c
2701             0.89 1.74 0.66 1.9 0.45 1.9 c
2702             0.3 1.9 0.0 1.81 0.0 1.42 c
2703             0.0 0.59 0.84 -0.11 1.96 -0.11 c
2704             3.21 -0.11 4.2 0.79 4.2 1.9 c
2705             h f
2706         }%
2707     \hfill
2708 }%
2709 }

2710 \tlconstcn{c_@@_glyph_6_tl}
2711 {%
2712     \hbox to 5pt
2713     {%
2714         \hfill
2715         \pdfextension literal
2716         {
2717             4.18 1.93 m
2718             4.18 3.07 3.3 3.96 2.2 3.96 c
2719             1.68 3.96 1.16 3.79 0.71 3.38 c
2720             0.85 4.79 1.79 5.61 2.67 5.61 c
2721             3.01 5.61 3.17 5.5 3.22 5.45 c
2722             3.17 5.4 3.12 5.34 3.12 5.16 c
2723             3.12 4.92 3.3 4.72 3.56 4.72 c
2724             3.81 4.72 4.01 4.89 4.01 5.19 c
2725             4.01 5.68 3.65 6.22 2.68 6.22 c
2726             1.34 6.22 0.0 5.01 0.0 2.99 c
2727             0.0 0.62 1.12 -0.11 2.11 -0.11 c
2728             3.2 -0.11 4.18 0.73 4.18 1.93 c
2729             3.49 1.93 m
2730             3.49 1.07 2.83 0.5 2.11 0.5 c
2731             1.46 0.5 1.07 0.99 0.87 1.6 c
2732             0.77 1.84 0.79 2.08 0.79 2.22 c
2733             0.79 2.84 1.39 3.34 2.15 3.34 c
2734             2.96 3.34 3.49 2.67 3.49 1.93 c
2735             h f
2736         }%

```

```

2737         \hfill
2738     }%
2739 }
2740 \tlconstcn{c_@@_glyph_7_tl}
2741 {%
2742     \hbox to 5pt
2743     {%
2744         \hfill
2745         \pdfextension literal
2746         {
2747             4.36 5.8 m
2748             4.36 6.11 4.12 6.11 3.96 6.11 c
2749             0.66 6.11 l
2750             0.6 6.27 0.42 6.27 0.34 6.27 c
2751             0.0 6.27 0.0 6.05 0.0 5.86 c
2752             0.0 5.44 l
2753             0.0 5.25 0.0 5.03 0.34 5.03 c
2754             0.69 5.03 0.69 5.22 0.69 5.5 c
2755             3.33 5.5 l
2756             1.6 3.66 1.26 1.46 1.26 0.36 c
2757             1.26 0.22 1.26 -0.11 1.61 -0.11 c
2758             1.78 -0.11 1.95 0.0 1.95 0.29 c
2759             2.0 3.13 3.53 4.87 4.14 5.46 c
2760             4.34 5.64 4.36 5.66 4.36 5.8 c
2761             h f
2762         }%
2763     \hfill
2764 }%
2765 }
2766 \tlconstcn{c_@@_glyph_8_tl}
2767 {%
2768     \hbox to 5pt
2769     {%
2770         \hfill
2771         \pdfextension literal
2772         {
2773             4.36 1.74 m
2774             4.36 2.49 3.71 3.08 2.97 3.29 c
2775             3.77 3.56 4.22 4.05 4.22 4.62 c
2776             4.22 5.44 3.37 6.22 2.18 6.22 c
2777             0.98 6.22 0.14 5.43 0.14 4.62 c
2778             0.14 4.05 0.6 3.55 1.39 3.29 c
2779             0.64 3.08 0.0 2.49 0.0 1.74 c
2780             0.0 0.77 0.93 -0.11 2.18 -0.11 c
2781             3.44 -0.11 4.36 0.77 4.36 1.74 c
2782             3.53 4.61 m
2783             3.53 4.04 2.91 3.6 2.18 3.6 c
2784             1.45 3.6 0.83 4.05 0.83 4.61 c
2785             0.83 5.13 1.39 5.61 2.18 5.61 c
2786             2.96 5.61 3.53 5.13 3.53 4.61 c
2787             3.67 1.75 m
2788             3.67 1.06 3.01 0.5 2.18 0.5 c
2789             1.35 0.5 0.69 1.06 0.69 1.75 c
2790             0.69 2.36 1.27 2.99 2.18 2.99 c
2791             3.1 2.99 3.67 2.36 3.67 1.75 c
2792             h f
2793         }%
2794     \hfill
2795 }%
2796 }
2797 \tlconstcn{c_@@_glyph_9_tl}
2798 {%
2799     \hbox to 5pt

```

```

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

```

2.16 We load piton.lua

```

2828 \cs_new_protected:Npn \@@_test_version:n #1
2829 {
2830     \str_if_eq:onF \PitonFileVersion { #1 }
2831     { \@@_error:n { bad~version~of~piton.lua } }
2832 }

2833 \hook_gput_code:nnn { begindocument } { . }
2834 {
2835     \lua_load_module:n { piton }
2836     \lua_now:n
2837     {
2838         tex.sprint ( luatexbase.catcodetables.expl ,
2839                     [[\@@_test_version:n {}] .. piton_version .. "]" )
2840     }
2841 }
</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`.

```

2842 ⟨*LUA⟩
2843 piton.comment_latex = piton.comment_latex or ">"
2844 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`.

```

2845 piton.write_files = { }
2846 piton.join_files = { }

2847 local sprintL3
2848 function sprintL3 ( s )
2849   tex.sprint ( luatexbase.catcodetables.expl , s )
2850 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.

```

2851 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2852 local Cg , Cmt , Cb = lpeg.Cg , lpeg.Cmt , lpeg.Cb
2853 local B , R = lpeg.B , lpeg.R

```

The following line is mandatory.

```

2854 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”).

```

2855 local Q
2856 function Q ( pattern )
2857   return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2858 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.

```

2859 local L
2860 function L ( pattern ) return
2861   Ct ( C ( pattern ) )
2862 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.

```

2863 local Lc
2864 function Lc ( string ) return
2865   Cc ( { luatexbase.catcodetables.expl , string } )
2866 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)

```

2867 local K
2868 function K ( style , pattern ) return
2869   Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
2870   * Q ( pattern )
2871   * Lc "}}"
2872 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.

```

2873 local WithStyle
2874 function WithStyle ( style , pattern ) return
2875   Ct ( Cc "Open" * Cc ( [[{\PitonStyle{ ]] .. style .. "}{" ) * Cc "}}" )
2876   * pattern
2877   * Ct ( Cc "Close" )
2878 end

```

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

```

2879 Escape = P ( false )
2880 EscapeClean = P ( false )
2881 if piton.begin_escape then
2882   Escape =
2883     P ( piton.begin_escape )
2884     * Lc [[\l_@@_after_begin_escape_tl]]
2885     * L ( ( 1 - P ( piton.end_escape ) ) ^ 0 )
2886     * Lc [[\l_@@_before_end_escape_tl]]
2887     * 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).

```

2888 EscapeClean =
2889   P ( piton.begin_escape )
2890   * ( 1 - P ( piton.end_escape ) ) ^ 0
2891   * P ( piton.end_escape )
2892 end
2893 EscapeMath = P ( false )
2894 if piton.begin_escape_math then
2895   EscapeMath =
2896     P ( piton.begin_escape_math )
2897     * Lc "$"
2898     * L ( ( 1 - P ( piton.end_escape_math ) ) ^ 1 )
2899     * Lc "$"
2900     * P ( piton.end_escape_math )
2901 end

```

The basic syntactic LPEG

```

2902 local alpha , digit = lpeg.alpha , lpeg.digit
2903 local space = P " "

```

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

```

2904 local letter = alpha + "_" + "â" + "ã" + "ç" + "ê" + "è" + "é" + "ë" + "ï" + "î"
2905               + "ô" + "û" + "ü" + "Â" + "Ã" + "Ç" + "É" + "Ê" + "Ë" + "Ï"
2906               + "Î" + "Ï" + "Ô" + "Õ" + "Û"
2907
2908 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).

```

2909 local identifier = letter * alphanum ^ 0

```

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

```

2910 local Identifier = K ( 'Identifier.Internal' , identifier )

```

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

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

```

2911 local allow_underscores_except_first
2912 function allow_underscores_except_first ( p )
2913     return p * ( P "_" + p ) ^ 0
2914 end
2915 local allow_underscores
2916 function allow_underscores ( p )
2917     return ( P "_" + p ) ^ 0
2918 end
2919 local digits_to_number
2920 function digits_to_number(prefix, digits)
2921     -- The edge cases of what is allowed in number literals is modelled after
2922     -- OCaml numbers, which seems to be the most permissive language
2923     -- in this regard (among C, OCaml, Python & SQL).
2924     return prefix
2925         * allow_underscores_except_first(digits^1)
2926         * ( P "." * #(1 - P ".") * allow_underscores(digits) ) ^ -1
2927         * ( S "eE" * S "+-" ^ -1 * allow_underscores_except_first(digits^1) ) ^ -1
2928 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).

```

2929 local Number =
2930     K ( 'Number.Internal' ,
2931         digits_to_number ( P "0x" + P "0X", R "af" + R "AF" + digit )
2932         + digits_to_number ( P "0o" + P "0O", R "07" )
2933         + digits_to_number ( P "0b" + P "0B", R "01" )
2934         + digits_to_number ( "" , digit )
2935     )

```

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?).

```

2936 local lpeg_central = 1 - S " '\r[({)}]" - digit

```

³The edge cases such as

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

```

2937 if piton.begin_escape then
2938   lpeg_central = lpeg_central - piton.begin_escape
2939 end
2940 if piton.begin_escape_math then
2941   lpeg_central = lpeg_central - piton.begin_escape_math
2942 end
2943 local Word = Q ( lpeg_central ^ 1 )

2944 local Space = Q " " ^ 1
2945 local SkipSpace = Q " " ^ 0
2946
2947 local Punct = Q ( S ".,:;!" )
2948
2949 local Tab = "\t" * Lc [[ \@_tab: ]]

2950 local LeadingSpace = Lc [[ \@_leading_space: ]] * P " "

2951 local Delim = K ( 'Delim' , S "[{ }]" )

```

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

```

2952 local SpaceInString = space * Lc [[ \l_@@_space_in_string_t1 ]]

```

3.3 The option 'detected-commands' and al.

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

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

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

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

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

```

2953 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
2954 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
2955 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
2956 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.

```

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

```

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

```

2961 local rawDetectedCommands = P ( false )
2962 for _ , x in ipairs ( raw_detected_commands ) do
2963   rawDetectedCommands = rawDetectedCommands + P ( "\\" .. x )
2964 end

```

```

2965 local beamerCommands = P ( false )
2966 for _ , x in ipairs ( beamer_commands ) do
2967     beamerCommands = beamerCommands + P ( "\\\" .. x )
2968 end
2969
2969 local beamerEnvironments = P ( false )
2970 for _ , x in ipairs ( beamer_environments ) do
2971     beamerEnvironments = beamerEnvironments + P ( x )
2972 end

```

Several tools for the construction of the main LPEG

```

2973 local LPEG0 = { }
2974 local LPEG1 = { }
2975 local LPEG2 = { }
2976 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*.

```

2977 local Compute_braces
2978 function Compute_braces ( lpeg_string ) return
2979     P { "E" ,
2980         E =
2981         (
2982             "{" * V "E" * "}"
2983             +
2984             lpeg_string
2985             +
2986             ( 1 - S "{" )
2987             ) ^ 0
2988     }
2989 end

```

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

```

2990 local Compute_DetectedCommands
2991 function Compute_DetectedCommands ( lang , braces ) return
2992     Ct (
2993         Cc "Open"
2994         * C ( detectedCommands * space ^ 0 * P "{" )
2995         * Cc "}"
2996     )
2997     * ( braces
2998         / ( function ( s )
2999             if s ~= '' then return
3000                 LPEG1[lang] : match ( s )
3001             end
3002             end )
3003     )
3004     * P "}"
3005     * Ct ( Cc "Close" )
3006 end
3007
3007 local Compute_RawDetectedCommands
3008 function Compute_RawDetectedCommands ( lang , braces ) return
3009     Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
3010 end

```

```

3011 local Compute_LPEG_cleaner
3012 function Compute_LPEG_cleaner ( lang , braces ) return
3013   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
3014         * ( braces
3015           / ( function ( s )
3016               if s ~= '' then return
3017                 LPEG_cleaner[lang] : match ( s )
3018             end
3019           end )
3020         )
3021       * "}"
3022       + EscapeClean
3023       + C ( P ( 1 ) )
3024       ) ^ 0 ) / table.concat
3025 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`.

```

3026 local ParseAgain
3027 function ParseAgain ( code )
3028   if code ~= '' then return

```

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

```

3029   LPEG1[piton.language] : match ( code )
3030 end
3031 end

```

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

```

3032 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.

```

3033 local Compute_Beamer
3034 function Compute_Beamer ( lang , braces )

```

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

```

3035   local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
3036   lpeg = lpeg +
3037     Ct ( Cc "Open"
3038         * C ( beamerCommands
3039             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3040             * P "{"
3041           )
3042         * Cc "}"
3043       )
3044     * ( braces /
3045         ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3046     * "}"
3047     * Ct ( Cc "Close" )

```

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

```

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

```

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

```

3056 lpeg = lpeg +
3057   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
3058   * ( braces
3059       / ( function ( s )
3060           if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3061   * L ( P "}" )
3062   * ( braces
3063       / ( function ( s )
3064           if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3065   * L ( P "}" )
3066   * ( braces
3067       / ( function ( s )
3068           if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
3069   * L ( P "}" )

```

Now, the environments of Beamer.

```

3070 for _ , x in ipairs ( beamer_environments ) do
3071   lpeg = lpeg +
3072     Ct ( Cc "Open"
3073         * C (
3074             P ( [[\begin{]] .. x .. "]" )
3075             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3076           )
3077             * space ^ 0 * ( P "\r" ) ^ 1
3078             * Cc ( [[\end{]] .. x .. "]" )
3079           )
3080         * (

```

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.

```

3081         (
3082           P { "E" ,
3083             E = (
3084               P ( [[\begin{]] .. x .. "]" )
3085               * V "E"
3086               * P ( [[\end{]] .. x .. "]" )
3087             +
3088               (
3089                 1
3090                 - P ( [[\begin{]] .. x .. "]" )
3091                 - P ( [[\end{]] .. x .. "]" )
3092               )
3093             ) ^ 0
3094           }
3095         )
3096       / ( function ( s )
3097           if s ~= '' then return
3098             LPEG1[lang] : match ( s )
3099           end
3100         end )
3101     )

```

```

3102         * P ( [[\end{}} .. x .. "]" )
3103         * Ct ( Cc "Close" )
3104     end

```

Now, you can return the value we have computed.

```

3105     return lpeg
3106 end

```

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

```

3107 local CommentMath =
3108     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`.

```

3109 local Prompt =
3110     K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
3111     * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

3112 local EOL =
3113     P "\r"
3114     *
3115     (
3116         space ^ 0 * -1
3117         +
3118         Cc "EOL"
3119     )
3120     *
3121     (
3122         Prompt -- added 2026-09-16 gh #55
3123         +
3124         ( LeadingSpace ^ 0 * # ( 1 - S "\r" ) ) ^ -1
3125     )

```

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

```

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

```

3.4 The language Python

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

```

3132 --python Python
3133 do

```

The following LPEG `expression` will be used for the interpolations in the Python strings but also for the parameters in the `argspec` of a Python function.

```

3134 local expression =
3135     P { "E" ,
3136         E = (
3137             "'" * ( P "\\'" + 1 - S "'\r" ) ^ 0 * "'"
3138             + "\" * ( P "\\\"" + 1 - S "\"\r" ) ^ 0 * "\""
3139             + "{" * V "F" * "}"
3140             + "(" * V "F" * ")"
3141             + "[" * V "F" * "]"

```

```

3141         + ( 1 - S "{ } ( ) [ ] \r , " ) ^ 0 ,
3142     F = (      "' ' * ( P "\\ ' " + 1 - S "' \r " ) ^ 0 * "' '
3143         + "\" \" * ( P "\\ \" \" \" + 1 - S "\" \r \" \" ) ^ 0 * "\" \"
3144         + "{ \" * V \"F\" * \"}\"
3145         + \"( \" * V \"F\" * \" ) \"
3146         + \"[ \" * V \"F\" * \" ] \"
3147         + ( 1 - S "{ } ( ) [ ] \r \" \" \" \" ) ^ 0
3148     }

```

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

```

3149     local Operator =
3150         K ( 'Operator' ,
3151             P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "/" + "*"
3152             + S "-~+/*%=<>&.@|" )
3153
3154     local OperatorWord =
3155         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.

```

3156     local For = K ( 'Keyword' , P "for" )
3157         * Space
3158         * Identifier
3159         * Space
3160         * K ( 'Keyword' , P "in" )
3161
3162     local Keyword =
3163         K ( 'Keyword' ,
3164             P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
3165             "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
3166             "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
3167             "try" + "type" + "while" + "with" + "yield" + "yield from" )
3168         + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
3169
3170     local Builtin =
3171         K ( 'Name.Builtin' ,
3172             P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
3173             "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
3174             "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
3175             "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
3176             "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
3177             "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
3178             + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
3179             "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
3180             "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" +
3181             "vars" + "zip" )
3182
3183     local Exception =
3184         K ( 'Exception' ,
3185             P "ArithmeticError" + "AssertionError" + "AttributeError" +
3186             "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
3187             "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
3188             "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
3189             "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
3190             "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
3191             "NotImplementedError" + "OSError" + "OverflowError" +
3192             "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
3193             "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
3194             "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
3195             + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
3196             "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
3197             "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +

```

```

3198     "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
3199     "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
3200     "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
3201     "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
3202     "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
3203     "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
3204     "RecursionError" )
3205
3206     local RaiseException =
3207     K ( 'Keyword' , P "raise" ) * SkipSpace * Exception * K ( 'Delim' , P "(" )

```

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

```

3208     local Decorator
3209     = K ( 'Name.Decorator' , P "@" * letter * ( alphanum + S "._" ) ^ 0 )

```

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:`

```

3210     local DefClass =
3211     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`

```

3212     local ImportAs =
3213     K ( 'Keyword' , "import" )
3214     * Space
3215     * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
3216     * (
3217         ( Space * K ( 'Keyword' , "as" ) * Space
3218           * K ( 'Name.Namespace' , identifier ) )
3219         +
3220         ( SkipSpace * Q "," * SkipSpace
3221           * K ( 'Name.Namespace' , identifier ) ) ^ 0
3222     )

```

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`

```

3223     local FromImport =
3224     K ( 'Keyword' , "from" )
3225     * Space * K ( 'Name.Namespace' , identifier )
3226     * Space * K ( 'Keyword' , "import" )

```

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

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

We have also to deal with the interpolations in the f-strings. Here is an example of a f-string with an interpolation and a format instruction⁴ in that interpolation:

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

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

```
3227 local PercentInterpol =
3228   K ( 'String.Interpol' ,
3229     P "%"
3230     * ( "(" * alphanum ^ 1 * ")" ) ^ -1
3231     * ( S "-#0 +" ) ^ 0
3232     * ( digit ^ 1 + "*" ) ^ -1
3233     * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
3234     * ( S "HLL" ) ^ -1
3235     * S "sdfFeExXorgiGauc%"
3236   )
```

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

```
3237 local SingleShortString =
3238   WithStyle ( 'String.Short.Internal' ,
```

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

```
3239   Q ( P "f'" + "F'" )
3240   * (
3241     K ( 'String.Interpol' , "{" )
3242     * K ( 'Interpol.Inside' , expression )
3243     * Q ( P ":" * ( 1 - S "}" ) ^ 0 ) ^ -1
3244     * K ( 'String.Interpol' , "}" )
3245     +
3246     SpaceInString
3247     +
3248     Q ( ( P "\\'" + "\\\\" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
3249   ) ^ 0
3250   * Q ""
3251   +
```

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

```
3252   Q ( P "'" + "r'" + "R'" )
3253   * ( Q ( ( P "\\'" + "\\\\" + 1 - S " '\r%" ) ^ 1 )
3254     + SpaceInString
3255     + PercentInterpol
3256     + Q "%"
3257   ) ^ 0
3258   * Q "" )
```

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

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

```

3259 local DoubleShortString =
3260     WithStyle ( 'String.Short.Internal' ,
3261         Q ( P "f\"" + "F\"" )
3262         * (
3263             K ( 'String.Interpol' , "{" )
3264             * K ( 'Interpol.Inside' , ( 1 - S "}" ) ^ 0 )
3265             * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}" ) ^ 0 ) ) ^ -1
3266             * K ( 'String.Interpol' , "}" )
3267             +
3268             SpaceInString
3269             +
3270             Q ( ( P "\\\"" + "\\\"" + "{{" + "}}" + 1 - S " {}\"" ) ^ 1 )
3271             ) ^ 0
3272         * Q "\""
3273     +
3274     Q ( P "\"" + "r\"" + "R\"" )
3275     * ( Q ( ( P "\\\"" + "\\\"" + 1 - S " \"r%" ) ^ 1 )
3276         + SpaceInString
3277         + PercentInterpol
3278         + Q "%"
3279         ) ^ 0
3280     * Q "\" )
3281
3282 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.

```

3283 local braces =
3284     Compute_braces
3285     (
3286         ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
3287         * ( P '\\"' + 1 - S "\" ) ^ 0 * "\"
3288     +
3289         ( P '\'' + 'r\'' + 'R\'' + 'f\'' + 'F\'' )
3290         * ( P '\\\'' + 1 - S '\'' ) ^ 0 * '\''
3291     )
3292
3293 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

Detected commands

```

3294 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
3295     + Compute_RawDetectedCommands ( 'python' , braces )

```

LPEG_cleaner

```

3296 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

The long strings

```

3297 local SingleLongString =
3298     WithStyle ( 'String.Long.Internal' ,
3299         ( Q ( S "fF" * P "'" )
3300         * (
3301             K ( 'String.Interpol' , "{" )
3302             * K ( 'Interpol.Inside' , ( 1 - S "}" - "'" ) ^ 0 )
3303             * Q ( P ":" * ( 1 - S "}" - "'" ) ^ 0 ) ^ -1
3304             * K ( 'String.Interpol' , "}" )

```

```

3305         +
3306         Q ( ( 1 - P "'''" - S "{}'\r" ) ^ 1 )
3307         +
3308         EOL
3309     ) ^ 0
3310 +
3311     Q ( ( S "rR" ) ^ -1 * "'''" )
3312     * (
3313         Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
3314         +
3315         PercentInterpol
3316         +
3317         P "%"
3318         +
3319         EOL
3320     ) ^ 0
3321 )
3322 * Q "'''" )
3323 local DoubleLongString =
3324     WithStyle ( 'String.Long.Internal' ,
3325     (
3326         Q ( S "fF" * "\"\"" )
3327         * (
3328             K ( 'String.Interpol', "{" )
3329             * K ( 'Interpol.Inside' , ( 1 - S "}:\" - "\"\"" ) ^ 0 )
3330             * Q ( ":" * (1 - S "}:\" - "\"\"" ) ^ 0 ) ^ -1
3331             * K ( 'String.Interpol' , "}" )
3332             +
3333             Q ( ( 1 - S "{}\\" - "\"\"" ) ^ 1 )
3334             +
3335             EOL
3336         ) ^ 0
3337     +
3338         Q ( S "rR" ^ -1 * "\"\"" )
3339         * (
3340             Q ( ( 1 - P "\"\"" - S "%\" ) ^ 1 )
3341             +
3342             PercentInterpol
3343             +
3344             P "%"
3345             +
3346             EOL
3347         ) ^ 0
3348     )
3349     * Q "\"\""
3350 )
3351 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).

```

3352 local StringDoc =
3353     K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\"" )
3354     * ( K ( 'String.Doc.Internal' , (1 - P "\"\"" - "\r" ) ^ 0 ) * EOL
3355         * Tab ^ 0
3356     ) ^ 0
3357     * 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.

```

3358 local Comment =

```

```

3359 WithStyle
3360   ( 'Comment.Internal' ,
3361     Q "#" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3362   )
3363   * ( EOL + -1 )

```

DefFunction 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`.

```

3364 local Params =
3365   P { "E" ,
3366     E = ( V "F" * ( Q "," * V "F" ) ^ 0 ) ^ -1 ,
3367     F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
3368       * ( Q ":" * SkipSpace * K ( 'Name.Type' , expression ) ) ^ -1
3369       * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
3370   }

```

The following LPEG DefFunction catches a keyword `def` and the following name of function *but also everything else until a potential docstring*. That's why this definition of LPEG must occur (in the file `piton.sty`) after the definition of several other LPEG such as `Comment`, `CommentLaTeX`, `Params`, `StringDoc`...

```

3371 local DefFunction =
3372   K ( 'Keyword' , "def" )
3373   * Space
3374   * K ( 'Name.Function.Internal' , identifier )
3375   * SkipSpace
3376   * K ( 'Delim' , P "(" ) * Params * K ( 'Delim' , P ")" )
3377   * SkipSpace
3378   * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3379   * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
3380   * Q ":"
3381   * ( SkipSpace
3382     * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
3383     * Tab ^ 0
3384     * SkipSpace
3385     * StringDoc ^ 0 -- there may be additional docstrings
3386   ) ^ -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

```

3387 local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

The main LPEG for the language Python

```
3388 local EndKeyword
3389     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3390     EscapeMath + -1
```

First, the main loop :

```
3391 local Main =
3392     space ^ 0 * EOL
3393     + Space
3394     + Tab
3395     + Escape + EscapeMath
3396     + Beamer
3397     + CommentLaTeX
3398     + DetectedCommands
3399     -- + Prompt -deleted 2026-09-16 : gh #55
3400     + LongString
3401     + Comment
3402     + ExceptionInConsole
3403     + Delim
3404     + Decorator -- position changed 2026-09-24 : gh #56
3405     + Operator
3406     + OperatorWord * EndKeyword
3407     + ShortString
3408     + Punct
3409     + FromImport
3410     + RaiseException
3411     + DefFunction
3412     + DefClass
3413     + For
3414     + Keyword * EndKeyword
3415     + Builtin * EndKeyword
3416     + Identifier
3417     + Number
3418     + Word
```

Here, we must not put local, of course.

```
3419 LPEG1.python = Main ^ 0
```

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

```
3420 LPEG2.python =
3421     Ct (
3422         ( space ^ 0 * "\r" ) ^ -1
3423         * Lc [[ \@@_begin_line: ]]
3424         * Prompt ^ -1 -- added 2026-09-16 gh #55
3425         * LeadingSpace ^ 0
3426         * ( space ^ 1 * -1 + Main ) ^ 0
3427         * -1
3428         * Lc [[ \@@_end_line: ]]
3429     )
```

End of the Lua scope for the language Python.

```
3430 end
```

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

3.5 The language OCaml

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

```

3431 --ocaml Ocaml OCaml
3432 do

3433     local SkipSpace = ( Q " " + EOL ) ^ 0
3434     local Space = ( Q " " + EOL ) ^ 1

3435     local braces = Compute_braces ( '\'' * ( 1 - S "\"" ) ^ 0 * '\'' )

3436     if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
3437     DetectedCommands =
3438         Compute_DetectedCommands ( 'ocaml' , braces )
3439         + Compute_RawDetectedCommands ( 'ocaml' , braces )
3440     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.

```

3441     function Q ( pattern, strict )
3442         if strict ~= nil then
3443             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3444         else
3445             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3446                 + Beamer + DetectedCommands + EscapeMath + Escape
3447         end
3448     end

3449     local K
3450     function K ( style , pattern, strict ) return
3451         Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
3452         * Q ( pattern, strict )
3453         * Lc "}"
3454     end

3455     local WithStyle
3456     function WithStyle ( style , pattern ) return
3457         Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{ ]] .. style .. "}{" ) * Cc "}" )
3458         * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
3459         * Ct ( Cc "Close" )
3460     end

```

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

```

3461     local balanced_parens =
3462         P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }

```

The strings of OCaml

```

3463     local ocaml_string =
3464         P "\""
3465         * (
3466             P " "
3467             +
3468             P ( ( P '\\\'' + 1 - S " \r" ) ^ 1 )
3469             +
3470             EOL -- ?
3471         ) ^ 0
3472     * P "\""

```

```

3473 local String =
3474   WithStyle
3475     ( 'String.Long.Internal' ,
3476       Q "\""
3477       * (
3478         SpaceInString
3479         +
3480         Q ( ( P '\\\\' + 1 - S "\r" ) ^ 1 )
3481         +
3482         EOL
3483       ) ^ 0
3484       * Q "\""
3485     )

```

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

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

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

```

3486 local ext = ( R "az" + "_" ) ^ 0
3487 local open = "{" * Cg ( ext , 'init' ) * "/"
3488 local close = "/" * C ( ext ) * "}"
3489 local closeeq =
3490   Cmt ( close * Cb ( 'init' ) ,
3491     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3492 local QuotedStringBis =
3493   WithStyle ( 'String.Long.Internal' ,
3494     (
3495       Space
3496       +
3497       Q ( ( 1 - S "\r" ) ^ 1 )
3498       +
3499       EOL
3500     ) ^ 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.

```

3501 local QuotedString =
3502   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3503   ( 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).

```

3504 local comment =
3505   P {
3506     "A" ,
3507     A = Q "(*"
3508       * ( V "A"
3509         + Q ( ( 1 - S "\r$\n" - "(*" - "*)" ) ^ 1 ) -- $
3510         + Q ( ocaml_string )
3511         + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3512         + EOL
3513       ) ^ 0
3514       * Q "*)"
3515   }
3516 local Comment = WithStyle ( 'Comment.Internal' , comment )

```

Some standard LPEG

```

3517 local Delim = K ( 'Delim' , P "[" + "]" + S "()" )
3518 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.

```

3519 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.

```

3520 local Constructor =
3521   P "::"

```

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

```

3522 * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3523 +
3524 P "[]"
3525 * Lc ([{\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]])
3526 K ( 'Name.Constructor' ,
3527     Q "`" ^ -1 * cap_identifier
3528     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )

```

```

3529 local ModuleType = K ( 'Name.Type' , cap_identifier )

```

```

3530 local OperatorWord =
3531   K ( 'Operator.Word' ,
3532       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )

```

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

```

3533 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3534   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3535   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3536   "struct" + "type" + "val"

```

```

3537 local Keyword =
3538   K ( 'Keyword' ,
3539       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3540       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3541       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3542       + "virtual" + "when" + "while" + "with" )
3543   + K ( 'Keyword.Constant' , P "true" + "false" )
3544   + K ( 'Keyword.Governing' , governing_keyword )

```

```

3545 local EndKeyword
3546   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3547   + EscapeMath + -1

```

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

```

3548 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3549   - ( 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`.

```

3550 local Identifier = K ( 'Identifier.Internal' , identifier )

```

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

```

3551 local ocaml_char =
3552   P "" *
3553   (
3554     ( 1 - S "\\\" )
3555     + "\\\"
3556     * ( S "\\ntbr \"
3557         + digit * digit * digit
3558         + P "x" * ( digit + R "af" + R "AF" )
3559                   * ( digit + R "af" + R "AF" )
3560                   * ( digit + R "af" + R "AF" )
3561         + P "o" * R "03" * R "07" * R "07" )
3562   )
3563   * ""
3564 local Char =
3565   K ( 'String.Short.Internal', ocaml_char )

```

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

```

3566 local TypeParameter =
3567   K ( 'TypeParameter' ,
3568       "" * 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.

```

3569 local DotNotation =
3570   ( K ( 'Name.Module' , cap_identifier )
3571     * Q "."
3572     * ( K ( 'Name.Module' , cap_identifier )
3573         * ( Q "." * K ( 'Name.Module' , cap_identifier ) ) ^ 0
3574         * ( Q "." * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ) ^ -1
3575         + Identifier + Constructor + Q "(" + Q "[" + Q "{" )
3576     + Identifier
3577     * Q "."
3578     * K ( 'Name.Field' , identifier ) )
3579   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0
3580   * ( Q "." * # Q "(" ) ^ -1

```

The records

```

3581 local expression_for_fields_type =
3582   P { "E" ,
3583       E = ( "{ * V "F" * }"
3584             + "(" * V "F" * ")"
3585             + TypeParameter
3586             + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
3587       F = ( "{ * V "F" * }"
3588             + "(" * V "F" * )"
3589             + ( 1 - S "{}()[]\r\"" ) + TypeParameter ) ^ 0
3590   }

```

```

3591 local expression_for_fields_value =
3592   P { "E" ,
3593       E = ( "{ * V "F" * }"
3594             + "(" * V "F" * )"
3595             + "[" * V "F" * "]"
3596             + ocaml_string + ocaml_char
3597             + ( 1 - S "{}()[];" ) ) ^ 0 ,
3598       F = ( "{ * V "F" * }"
3599             + "(" * V "F" * )"
3600             + "[" * V "F" * "]"
3601             + ocaml_string + ocaml_char
3602             + ( 1 - S "{}()[]\\"" ) ) ^ 0
3603   }

```

```

3604 local OneFieldDefinition =
3605   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3606   * K ( 'Name.Field' , identifier ) * SkipSpace
3607   * Q ":" * SkipSpace
3608   * K ( 'TypeExpression' , expression_for_fields_type )
3609   * SkipSpace

```

```

3610 local OneField =
3611   K ( 'Name.Field' , identifier ) * SkipSpace

```

A value field may use OCaml's abbreviated syntax: { field } means { field = field }.

```

3612   * (
3613     Q "=" * SkipSpace
3614     * ( C ( expression_for_fields_value ) / ParseAgain )
3615     * SkipSpace
3616     + P ""
3617   )

```

The records.

```

3618 local RecordVal =
3619   Q "{" * SkipSpace
3620   *
3621   (
3622     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3623   ) ^ -1
3624   *
3625   (
3626     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3627   )
3628   * SkipSpace
3629   * Q ";" ^ -1
3630   * SkipSpace
3631   * Comment ^ -1
3632   * SkipSpace
3633   * Q "}"
3634 local RecordType =
3635   Q "{" * SkipSpace
3636   *
3637   (
3638     OneFieldDefinition
3639     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3640   )
3641   * SkipSpace
3642   * Q ";" ^ -1
3643   * SkipSpace
3644   * Comment ^ -1
3645   * SkipSpace
3646   * Q "}"
3647 local Record = RecordType + RecordVal

```

```

3648 local Operator =
3649   P "||" *

```

Don't use \hspace instead of \kern!

```

3650 Lc([[{\PitonStyle{Operator}}{\kern0.1em/\kern-0.2em/\kern0.1em}}]])
3651 +
3652 K ( 'Operator' ,
3653   P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3654   "/" + "*" + ";" + "->" + "+" + "-" + "." + "/" +
3655   + S "--+/*%=<>&@|" )

```

```

3656 local Builtin =
3657   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

3658 local Exception =
3659   K ( 'Exception' ,
3660     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3661     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3662     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

3663 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).

```

3664 local pattern_part =
3665   ( 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).

```
3666 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.

```

3667   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3668   *

```

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

```

3669   (
3670     K ( 'Identifier.Internal' , identifier )
3671     +
3672     Q "(" * SkipSpace
3673     * ( C ( pattern_part ) / ParseAgain )
3674     * SkipSpace

```

Of course, the specification of type is optional.

```

3675     * ( Q ":@" * #(1- P"=")
3676         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3677         ) ^ -1
3678     * Q ")"
3679   )

```

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

```

3680 local DefFunction =
3681   K ( 'Keyword.Governing' , "let open" )
3682   * Space
3683   * K ( 'Name.Module' , cap_identifier )
3684   +
3685   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3686   * Space
3687   * K ( 'Name.Function.Internal' , identifier )
3688   * Space
3689   * (

```

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).

```

3690      Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3691      +
3692      Argument * ( SkipSpace * Argument ) ^ 0
3693      * (
3694          SkipSpace
3695          * Q ":" * # ( 1 - P "=" )
3696          * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3697      ) ^ -1
3698  )

```

DefModule

```

3699  local DefModule =
3700      K ( 'Keyword.Governing' , "module" ) * Space
3701      *
3702      (
3703          K ( 'Keyword.Governing' , "type" ) * Space
3704          * K ( 'Name.Type' , cap_identifier )
3705      +
3706      K ( 'Name.Module' , cap_identifier ) * SkipSpace
3707      *
3708      (
3709          Q "(" * SkipSpace
3710          * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3711          * Q ":" * # ( 1 - P "=" ) * SkipSpace
3712          * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3713          *
3714          (
3715              Q "," * SkipSpace
3716              * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3717              * Q ":" * # ( 1 - P "=" ) * SkipSpace
3718              * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3719          ) ^ 0
3720          * Q ")"
3721      ) ^ -1
3722      *
3723      (
3724          Q "=" * SkipSpace
3725          * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3726          * Q "("
3727          * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3728          *
3729          (
3730              Q ","
3731              *
3732              K ( 'Name.Module' , cap_identifier ) * SkipSpace
3733          ) ^ 0
3734          * Q ")"
3735      ) ^ -1
3736      )
3737      +
3738      K ( 'Keyword.Governing' , P "include" + "open" )
3739      * Space
3740      * K ( 'Name.Module' , cap_identifier )

```

DefType

```

3741  local DefType =
3742      K ( 'Keyword.Governing' , "type" )
3743      * Space
3744      * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3745      * SkipSpace

```

```

3746 * ( Q "+=" + Q "=" )
3747 * SkipSpace
3748 * (
3749     RecordType
3750     +

```

The following lines are a suggestion of Y. Salmon.

```

3751     WithStyle
3752     (
3753         'TypeExpression' ,
3754         (
3755             (
3756                 EOL
3757                 + comment
3758                 + Q ( 1
3759                     - P ";;"
3760                     - P "type"
3761                     - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3762                 )
3763             ) ^ 0
3764             *
3765             (
3766                 # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3767                 + Q ";;"
3768                 + -1
3769             )
3770         )
3771     )
3772 )

3773 local prompt =
3774   Q "utop[" * digit^1 * Q "]"> "
3775 local start_of_line = P(function(subject, position)
3776   if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3777     return position
3778   end
3779   return nil
3780 end)
3781 local Prompt = #start_of_line * K( 'Prompt', prompt )
3782 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3783               * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3784               * ( K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

The main LPEG for the language OCaml

```

3785 local Main =
3786   space ^ 0 * EOL
3787   + Space
3788   + Tab
3789   + Escape + EscapeMath
3790   + Beamer
3791   + DetectedCommands
3792   + TypeParameter
3793   + String + QuotedString + Char
3794   + Comment
3795   + Prompt + Answer

```

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

```

3796   + Q "~" * Identifier * ( Q ":" ) ^ -1
3797   + Q ":" * # ( 1 - P ":" ) * SkipSpace
3798     * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3799   + Exception
3800   + DefType

```

```

3801      + DefFunction
3802      + DefModule
3803      + Record
3804      + Keyword * EndKeyword
3805      + OperatorWord * EndKeyword
3806      + Builtin * EndKeyword
3807      + DotNotation * EndKeyword
3808      + Constructor
3809      + Identifier
3810      + Punct
3811      + Delim -- Delim is before Operator for a correct analysis of [| et |]
3812      + Operator
3813      + Number
3814      + Word

```

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

```

3815      LPEG1.ocaml = Main ^ 0

```

```

3816      LPEG2.ocaml =
3817      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).

```

3818      (
3819      (
3820          P ":"
3821          +
3822          (
3823              ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3824              * Identifier
3825              * SkipSpace
3826              * Q ":"
3827          )
3828      )
3829      * # ( 1 - S ":@" )
3830      * SkipSpace
3831      * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 ) * -1
3832      )
3833      +
3834      (
3835          ( space ^ 0 * "\r" ) ^ -1
3836          * Lc [[ \@@_begin_line: ]]
3837          * LeadingSpace ^ 0
3838          * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
3839              + space ^ 0 * EOL
3840              + Main
3841          ) ^ 0
3842          * -1
3843          * Lc [[ \@@_end_line: ]]
3844      )
3845      )

```

End of the Lua scope for the language OCaml.

```

3846      end

```

3.6 The language C

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

```
3847 --c C c++ C++
3848 do
```

```
3849     local Delim = K ( 'Delim' , S "{[()]}" )
3850     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.

```
3851     local identifier = letter * alphanum ^ 0
3852
3853     local Operator =
3854         K ( 'Operator' ,
3855             P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3856             + S "-~/*%=<>.&|!" )
3857
3858     local Keyword =
3859         K ( 'Keyword' ,
3860             P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3861             "constexpr" + "const" + "continue" + "decltype" + "do" + "else" + "enum" +
3862             "extern" + "for" + "goto" + "if" + "noexcept" + "private" + "public" +
3863             "register" + "restrict" + "return" + "static_assert" + "static" + "struct" +
3864             "switch" + "thread_local" + "throw" + "try" + "typedef" + "union" +
3865             "using" + "virtual" + "volatile" + "while"
3866         )
3867         + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3868
3869     local Builtin =
3870         K ( 'Name.Builtin' ,
3871             P "alignof" + "free" + "malloc" + "printf" + "scanf" + "sizeof" )
3872
3873     local Type =
3874         K ( 'Name.Type' ,
3875             P "FILE" + "bool" + "char" + "char16_t" + "char32_t" + "clock_t" +
3876             "double" + "float" + "fpos_t" +
3877             "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3878             "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3879             "ptrdiff_t" + "size_t" + "time_t" + "void" + "wchar_t" )
3880     * -alphanum * Space ^ 0 * Operator ^ 0
```

The keywords `struct`, `union` and `enum` are formatted as keywords. When one of them is followed by an identifier, that identifier is formatted as a type. The pattern `TypeSpecifier` groups the predefined types and the qualifiers which may occur in a `typedef` declaration.

```
3881
3882     local AggregateKeyword = K ( 'Keyword' , P "struct" + "union" + "enum" )
3883     local TypeQualifier =
3884         K ( 'Keyword' , P "const" + "volatile" ) * Space ^ 0
3885     local TypeSpecifier = Type + TypeQualifier
3886     local AggregateName =
3887         AggregateKeyword * Space
3888         * K ( 'Name.Type' , identifier ) * Space ^ 0
3889
3890     local DefEnumClass =
3891         K ( 'Keyword' , P "enum" ) * Space * K ( 'Keyword' , P "class" ) * Space
3892         * K ( 'Name.Type' , identifier ) * Space ^ 0
```

To parse a `typedef` declaration containing an aggregate body, we first find the matching closing brace. Strings, character literals and comments are considered atomic so that a closing brace within one of them is ignored.

```

3893
3894 local aggregate_string = P "\"" * ( P "\\\"" + 1 - S "\"" ) ^ 0 * P "\""
3895 local aggregate_character =
3896   P "'" * ( P "\\'" + P "\\\"" + 1 - S "'" ) ^ 0 * P "'"
3897 local aggregate_comment =
3898   P "//" * ( 1 - P "\r" ) ^ 0 + P "/*" * ( 1 - P "*/" ) ^ 0 * P "*/"
3899 local aggregate_braces =
3900   Compute_braces ( aggregate_string + aggregate_character + aggregate_comment )
3901 local AggregateBraces = C ( P "{" * aggregate_braces * P "}" )
3902

```

The aggregate body is parsed again with the main lexer. Hence its line breaks and its internal syntax highlighting are preserved while the identifier created by `typedef` is formatted as `Name.Type`.

```

3903
3904 local DefTypedefAggregate =
3905   K ( 'Keyword' , P "typedef" ) * Space * AggregateKeyword * SkipSpace
3906   * ( K ( 'Name.Type' , identifier ) * SkipSpace ) ^ -1
3907   * ( AggregateBraces / ParseAgain ) * SkipSpace
3908   * ( Operator * SkipSpace ) ^ 0
3909   * K ( 'Name.Type' , identifier ) * Q ";"
3910
3911 local DefTypedef =
3912   K ( 'Keyword' , P "typedef" ) * Space
3913   * (
3914     TypeSpecifier * TypeSpecifier ^ 0 * Space ^ 0
3915     + AggregateName * Operator ^ 0 * Space ^ 0
3916   )
3917   * K ( 'Name.Type' , identifier )
3918   * Q ";"
3919

```

The following patterns handle declarations beginning with a user type. Since `piton` does not maintain any symbol table, a pointer declaration of the form `Type * variable` can not always be distinguished from an expression of the form `value * value`. These patterns therefore require a declaration boundary before the type, as well as a declaration-like suffix. Preprocessor directives are ignored when looking for this boundary: unlike an ordinary C statement, a directive need not end with a semicolon.

```

3920
3921 local SkipCWhitespace = ( space + P "\r" + P "\t" ) ^ 0
3922 local function PreviousCCharacter ( s , pos )
3923   local i = pos - 1
3924   while i > 0 do
3925     while i > 0 and s : sub ( i , i ) : match ( "%s" ) do i = i - 1 end
3926     if i == 0 then return nil end
3927     if i > 1 and s : sub ( i - 1 , i ) == "*/" then
3928       local j = i - 2
3929       while j > 1 and s : sub ( j - 1 , j ) ~= "/*" do j = j - 1 end
3930       if j > 1 then i = j - 2
3931       else return s : sub ( i , i ) , i end
3932     else
3933       local line_start = i
3934       while line_start > 0 and s : sub ( line_start , line_start ) ~= "\r" do
3935         line_start = line_start - 1
3936       end
3937       local line = s : sub ( line_start + 1 , i )
3938       if line : match ( "[\t]*##" ) then
3939         -- A preprocessor directive is a declaration boundary.
3940         i = line_start
3941       else
3942         local comment = line : find ( "//" , 1 , true )
3943         if comment then
3944           local before_comment = line : sub ( 1 , comment - 1 )

```

```

3945         if before_comment : match ( "%s*$" ) or before_comment : match ( "%s$" ) then
3946             i = line_start + comment - 1
3947         else
3948             return s : sub ( i , i ) , i
3949         end
3950     else
3951         return s : sub ( i , i ) , i
3952     end
3953 end
3954 end
3955 end
3956 return nil
3957 end
3958
3959 local DeclarationStart = Cmt ( P ( true ) ,
3960     function ( s , pos )
3961         local previous = PreviousCCharacter ( s , pos )
3962         if previous == nil or previous == ";" or previous == "}" or previous == "{" then
3963             return pos
3964         end
3965     end
3966 )
3967
3968 local FunctionParameters = P {
3969     "Parameters" ,
3970     Parameters = P "("
3971         * ( aggregate_string + aggregate_character + aggregate_comment
3972             + V "Parameters" + ( 1 - S "(" ) )
3973         ) ^ 0
3974         * P ")"
3975 }
3976
3977 local FunctionParameterStart = Cmt ( P ( true ) ,
3978     function ( s , pos )
3979         local separator , i = PreviousCCharacter ( s , pos )
3980         if separator ~= "(" and separator ~= "," then return end
3981
3982         local depth , open = 0 , nil
3983         local j = i
3984         while j > 0 do
3985             local c , at = PreviousCCharacter ( s , j + 1 )
3986             if not c then break end
3987             if c == ")" then depth = depth + 1
3988             elseif c == "(" then
3989                 if depth == 0 then open = at ; break end
3990                 depth = depth - 1
3991             end
3992             j = at - 1
3993         end
3994         if not open then return end
3995
3996         local boundary = open - 1
3997         while boundary > 0 do
3998             local c , at = PreviousCCharacter ( s , boundary + 1 )
3999             if not c then boundary = 0 ; break end
4000             if c == ";" or c == "{" or c == "}" then boundary = at ; break end
4001             boundary = at - 1
4002         end
4003         local header = s : sub ( boundary + 1 , open - 1 )
4004             : gsub ( "%*.*%/" , " " )
4005             : gsub ( "//[^\r]*" , " " )
4006             : gsub ( "[^\t]*#[^\r]*" , " " )
4007             : gsub ( "\r[^\t]*#[^\r]*" , "\r" )

```

```

4008         if header : find ( "[~%w_%s%*]" ) then return end
4009         local words = 0
4010         for _ in header : gmatch ( "[%a_][%w_]*" ) do words = words + 1 end
4011         if words < 2 then return end
4012         return pos
4013     end
4014 )
4015
4016 local UserPointerDeclarator =
4017     identifier * SkipSpace * P "*" * SkipSpace * identifier
4018     * # ( SkipCWhitespace * ( P ";" + P "=" + P "," + P ")" ) )
4019
4020 local UserTypePointerVariable =
4021     ( DeclarationStart + # UserPointerDeclarator * FunctionParameterStart )
4022     * K ( 'Name.Type' , identifier ) * SkipSpace
4023     * K ( 'Operator' , P "*" ) * SkipSpace
4024     * Identifier
4025     * # ( SkipCWhitespace * P ";"
4026           + SkipCWhitespace * P "=" * # ( 1 - P "=" )
4027           + SkipCWhitespace * ( P "," + P ")" )
4028     )
4029
4030 local UserTypeVariable =
4031     K ( 'Name.Type' , identifier ) * Space * Identifier
4032     * # ( Punct + Delim + Space * P "=" + EOL + -1 )

```

The LPEG DefFunction deals with the definitions and the prototypes of the functions. The returned type may be a predefined type, an aggregate name (e.g. `struct Point * f()`) or a user-defined type, followed by any number of stars. A declaration boundary and a complete parameter list ending in `;` or `{` are required, to avoid treating an expression such as `result = value * function(...)` as a function declaration.

```

4033
4034 local ReturnTypes =
4035     ( Type + AggregateName + K ( 'Name.Type' , identifier ) )
4036     * ( Space ^ 0 * K ( 'Operator' , P "*" ) ) ^ 0
4037     * Space ^ 0
4038
4039 local DefFunction =
4040     DeclarationStart
4041     * ReturnTypes
4042     * K ( 'Name.Function.Internal' , identifier )
4043     * SkipSpace
4044     * # ( FunctionParameters * SkipCWhitespace * ( P ";" + 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 python style `Name.Class`).

Example: `class myclass:`

```

4045 local DefClass =
4046     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).

```

4047 local Character =
4048     K ( 'String.Short' ,
4049         P [['\']] + P "'" * ( 1 - P "'" ) ^ 0 * P "'" )

```

The strings of C

```
4050 String =
4051   WithStyle ( 'String.Long.Internal' ,
4052     Q "\""
4053     * ( SpaceInString
4054       + K ( 'String.Interpol' ,
4055         "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
4056       )
4057       + Q ( ( P "\\\"" + 1 - S " \" " ) ^ 1 )
4058     ) ^ 0
4059     * Q "\""
4060   )
```

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

```
4061 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
4062 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end
4063 DetectedCommands =
4064   Compute_DetectedCommands ( 'c' , braces )
4065   + Compute_RawDetectedCommands ( 'c' , braces )
4066 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )
```

The directives of the preprocessor

```
4067 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.

```
4068 local Comment =
4069   WithStyle ( 'Comment.Internal' ,
4070     Q "//" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
4071     * ( EOL + -1 )
4072
4073 local LongComment =
4074   WithStyle ( 'Comment.Internal' ,
4075     Q "/*"
4076     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
4077     * Q "*/"
4078   ) -- $
```

The main LPEG for the language C

```
4079 local EndKeyword
4080   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
4081   EscapeMath + -1
```

First, the main loop :

```
4082 local Main =
4083   space ^ 0 * EOL
4084   + Space
4085   + Tab
4086   + Escape + EscapeMath
4087   + CommentLaTeX
4088   + Beamer
4089   + DetectedCommands
4090   + Preproc
```

```

4091      + Comment + LongComment
4092      + Delim
4093      + Operator
4094      + Character
4095      + String
4096      + Punct
4097      + DefTypedefAggregate
4098      + DefTypedef
4099      + DefEnumClass
4100      + DefFunction
4101      + AggregateName
4102      + DefClass
4103      + Type
4104      + Keyword * EndKeyword
4105      + UserTypePointerVariable
4106      + UserTypeVariable
4107      + Builtin * EndKeyword
4108      + Identifier
4109      + Number
4110      + Word

```

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

```

4111   LPEG1.c = Main ^ 0

```

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

```

4112   LPEG2.c =
4113   Ct (
4114       ( space ^ 0 * P "\r" ) ^ -1
4115       * Lc [[ \@@_begin_line: ]]
4116       * LeadingSpace ^ 0
4117       * ( space ^ 1 * -1 + Main ) ^ 0
4118       * -1
4119       * Lc [[ \@@_end_line: ]]
4120   )

```

End of the Lua scope for the language C.

```

4121 end

```

3.7 The language SQL

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

```

4122 --sql SQL
4123 do

4124   local LuaKeyword
4125   function LuaKeyword ( name ) return
4126       Lc [[ {\PitonStyle{Keyword}{ } ]]
4127       * Q ( Cmt (
4128           C ( letter * alphanum ^ 0 ) ,
4129           function ( _ , _ , a ) return a : upper ( ) == name end
4130       )
4131   )
4132   * Lc "}}"
4133 end

```

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

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

```

4134 local identifier =
4135     letter * ( alphanum + "-" ) ^ 0
4136     + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"

4137 local Operator =
4138     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).

```

4139 local Set
4140 function Set ( list )
4141     local set = { }
4142     for _ , l in ipairs ( list ) do set[l] = true end
4143     return set
4144 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.

```

4145 local set_keywords = Set
4146 {
4147     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
4148     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
4149     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
4150     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
4151     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
4152     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
4153     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
4154     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
4155     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
4156     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
4157     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
4158     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
4159     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
4160     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
4161     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
4162     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
4163     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
4164     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
4165     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
4166     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
4167 }

4168 local set_builtins = Set
4169 {
4170     "AVG" , "COUNT" , "CHAR_LENGTH" , "CONCAT" , "CURDATE" , "CURRENT_DATE" ,
4171     "DATE_FORMAT" , "DAY" , "LOWER" , "LTRIM" , "MAX" , "MIN" , "MONTH" , "NOW" ,
4172     "RANK" , "ROUND" , "RTRIM" , "SUBSTRING" , "SUM" , "UPPER" , "YEAR"
4173 }

```

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.

```

4174 local Identifier =
4175     C ( identifier ) /
4176     (
4177         function ( s )
4178             if set_keywords [ s : upper ( ) ] then return

```

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

```

4179     { [[{\PitonStyle{Keyword}{}}] } ,
4180     { luatexbase.catcodetables.other , s } ,
4181     { "}" }
4182   else
4183     if set_builtins [ s : upper ( ) ] then return
4184       { [[{\PitonStyle{Name.Builtin}{}}] } ,
4185       { luatexbase.catcodetables.other , s } ,
4186       { "}" }
4187     else return
4188       { [[{\PitonStyle{Name.Field}{}}] } ,
4189       { luatexbase.catcodetables.other , s } ,
4190       { "}" }
4191     end
4192   end
4193 end
4194 )

```

The strings of SQL

```

4195 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.

```

4196 local braces = Compute_braces ( "" * ( 1 - P "" ) ^ 1 * "" )
4197 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
4198 DetectedCommands =
4199   Compute_DetectedCommands ( 'sql' , braces )
4200   + Compute_RawDetectedCommands ( 'sql' , braces )
4201 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.

```

4202 local Comment =
4203   WithStyle ( 'Comment.Internal' ,
4204     Q "--" -- syntax of SQL92
4205     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
4206     * ( EOL + -1 )
4207
4208 local LongComment =
4209   WithStyle ( 'Comment.Internal' ,
4210     Q "/*"
4211     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
4212     * Q "*/"
4213     ) -- $

```

The main LPEG for the language SQL

```

4214 local EndKeyword
4215   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
4216     EscapeMath + -1

```

```

4217 local TableField =
4218     K ( 'Name.Table' , identifier )
4219     * Q "."
4220     * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
4221
4222 local OneField =
4223     (
4224         Q ( "(" * ( 1 - P ) ) ^ 0 * ")" )
4225         +
4226         K ( 'Name.Table' , identifier )
4227         * Q "."
4228         * K ( 'Name.Field' , identifier )
4229         +
4230         K ( 'Name.Field' , identifier )
4231     )
4232     * (
4233         Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
4234     ) ^ -1
4235     * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
4236
4237 local OneTable =
4238     K ( 'Name.Table' , identifier )
4239     * (
4240         Space
4241         * LuaKeyword "AS"
4242         * Space
4243         * K ( 'Name.Table' , identifier )
4244     ) ^ -1
4245
4246 local WeCatchTableNames =
4247     LuaKeyword "FROM"
4248     * ( Space + EOL )
4249     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
4250     + (
4251         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
4252         + LuaKeyword "TABLE"
4253     )
4254     * ( Space + EOL ) * OneTable
4255
4256 local EndKeyword
4257     = Space + Punct + Delim + EOL + Beamer
4258     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

4258 local Main =
4259     space ^ 0 * EOL
4260     + Space
4261     + Tab
4262     + Escape + EscapeMath
4263     + CommentLaTeX
4264     + Beamer
4265     + DetectedCommands
4266     + Comment + LongComment
4267     + Delim
4268     + Operator
4269     + String
4270     + Punct
4271     + WeCatchTableNames
4272     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
4273     + Number
4274     + Word

```

Here, we must not put local, of course.

```

4275 LPEG1.sql = Main ^ 0

```

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

```

4276   LPEG2.sql =
4277     Ct (
4278       ( space ^ 0 * "\r" ) ^ -1
4279       * Lc [[ \@@_begin_line: ]]
4280       * LeadingSpace ^ 0
4281       * ( space ^ 1 * -1 + Main ) ^ 0
4282       * -1
4283       * Lc [[ \@@_end_line: ]]
4284     )

```

End of the Lua scope for the language SQL.

```

4285 end

```

3.8 The language “Minimal”

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

```

4286 --minimal Minimal
4287 do
4288   local Punct = Q ( S " , ; ! \ " )
4289
4290   local Comment =
4291     WithStyle ( 'Comment.Internal' ,
4292       Q "#"
4293       * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4294     )
4295     * ( EOL + -1 )
4296
4297   local String =
4298     WithStyle ( 'String.Short.Internal' ,
4299       Q "\""
4300       * ( SpaceInString
4301         + Q ( ( P "[\]" ) + 1 - S " \"" ) ^ 1 )
4302       ) ^ 0
4303       * Q "\""
4304     )

```

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

```

4305   local braces = Compute_braces ( P "\"" * ( P "\\\"" + 1 - P "\"" ) ^ 1 * "\"" )
4306
4307   if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
4308
4309   DetectedCommands =
4310     Compute_DetectedCommands ( 'minimal' , braces )
4311     + Compute_RawDetectedCommands ( 'minimal' , braces )
4312
4313   LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
4314
4315   local identifier = letter * alphanum ^ 0
4316
4317   local Identifier = K ( 'Identifier.Internal' , identifier )
4318
4319   local Delim = Q ( S "{[()]}" )
4320
4321   local Main =

```

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

```

4322     space ^ 0 * EOL
4323     + Space
4324     + Tab
4325     + Escape + EscapeMath
4326     + CommentLaTeX
4327     + Beamer
4328     + DetectedCommands
4329     + Comment
4330     + Delim
4331     + String
4332     + Punct
4333     + Identifier
4334     + Number
4335     + Word

```

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

```

4336     LPEG1.minimal = Main ^ 0
4337
4338     LPEG2.minimal =
4339     Ct (
4340         ( space ^ 0 * "\r" ) ^ -1
4341         * Lc [[ \@@_begin_line: ]]
4342         * LeadingSpace ^ 0
4343         * ( space ^ 1 * -1 + Main ) ^ 0
4344         * -1
4345         * Lc [[ \@@_end_line: ]]
4346     )

```

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

```

4347 end

```

3.9 The language “Verbatim”

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

```

4348 --verbatim Verbatim
4349 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”).

```

4350     local braces =
4351     P { "E" ,
4352         E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4353     }
4354
4355     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
4356
4357     DetectedCommands =
4358     Compute_DetectedCommands ( 'verbatim' , braces )
4359     + Compute_RawDetectedCommands ( 'verbatim' , braces )
4360
4361     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

4362     local lpeg_central = 1 - S " \\r"
4363     if piton.begin_escape then
4364         lpeg_central = lpeg_central - piton.begin_escape
4365     end
4366     if piton.begin_escape_math then
4367         lpeg_central = lpeg_central - piton.begin_escape_math
4368     end
4369     local Word = Q ( lpeg_central ^ 1 )

```

```

4370
4371 local Main =
4372     space ^ 0 * EOL
4373     + Space
4374     + Tab
4375     + Escape + EscapeMath
4376     + Beamer
4377     + DetectedCommands
4378     + Q [[\]]
4379     + Word

```

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

```

4380 LPEG1.verbatim = Main ^ 0
4381
4382 LPEG2.verbatim =
4383     Ct (
4384         ( space ^ 0 * "\r" ) ^ -1
4385         * Lc [[ \@@_begin_line: ]]
4386         * LeadingSpace ^ 0
4387         * ( space ^ 1 * -1 + Main ) ^ 0
4388         * -1
4389         * Lc [[ \@@_end_line: ]]
4390     )

```

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

```

4391 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).

```

4392 --EXPL expl
4393 do
4394     local Comment =
4395         WithStyle
4396         ( 'Comment.Internal' ,
4397         Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4398         )
4399     * ( EOL + -1 )

```

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

```

4400 local analyze_cs
4401 function analyze_cs ( s )
4402     local i = s : find ( ":" )
4403     if i then

```

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

```

4404     local name = s : sub ( 2 , i - 1 )
4405     local parts = name : explode ( "_" )
4406     local module = parts[1]
4407     if module == "" then module = parts[3] end

```

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

```

4408     return
4409     { [[{\OptionalLocalPitonStyle{Module.}} .. module .. "}{"} } ,
4410     { luatexbase.catcodetables.other , s } ,
4411     { "}" } }
4412 else

```

```

4413     local p = s : sub ( 1 , 3 )
4414     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`.

```

4415         local scope = s : sub(2,2)
4416         local parts = s : explode ( "_" )
4417         local module = parts[2]
4418         if module == "" then module = parts[3] end
4419         local type = parts[#parts]
4420         return
4421         { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
4422           { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
4423           { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,
4424           { luatexbase.catcodetables.other , s } ,
4425           { "}}}}}}" }
4426     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.

```

4427         return { luatexbase.catcodetables.other , s }
4428     end
4429 end
4430 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`).

```

4431     local braces =
4432     P { "E" ,
4433       E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
4434     }
4435
4436     if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
4437
4438     DetectedCommands =
4439     Compute_DetectedCommands ( 'expl' , braces )
4440     + Compute_RawDetectedCommands ( 'expl' , braces )
4441
4442     LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
4443
4444     local control_sequence = P "\\\" * ( R "Az" + "_" + ":" + "@" ) ^ 1
4445     local ControlSequence = C ( control_sequence ) / analyze_cs
4446
4447     local def_function
4448     = P [[\cs_]]
4449     * ( P "set" + "new" )
4450     * ( P "_protected" ) ^ -1
4451     * P ":N" * ( P "p" ) ^ -1 * "n"
4452
4453     local DefFunction =
4454     C ( def_function ) / analyze_cs
4455     * Space
4456     * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
4457     * ControlSequence -- Q ( ControlSequence ) ?
4458     * Lc "}"
4459
4460     local Word = Q ( ( 1 - S " \r" ) ^ 1 )
4461
4462     local Main =
4463     space ^ 0 * EOL
4464     + Space
4465     + Tab
4466     + Escape + EscapeMath
4467     + Beamer
4468     + Comment
4469     + DetectedCommands
4470     + DefFunction

```

```

4467     + ControlSequence
4468     + Word

```

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

```

4469   LPEG1.expl = Main ^ 0
4470
4471   LPEG2.expl =
4472     Ct (
4473       ( space ^ 0 * "\r" ) ^ -1
4474       * Lc [[ \@@_begin_line: ]]
4475       * LeadingSpace ^ 0
4476       * ( space ^ 1 * -1 + Main ) ^ 0
4477       * -1
4478       * Lc [[ \@@_end_line: ]]
4479     )

```

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

```

4480 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.

```

4481 function piton.Parse ( language , code )

```

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

```

4482   piton.language = language
4483   local t = LPEG2[language] : match ( code )
4484   if not t then
4485     sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
4486     return -- to exit in force the function
4487   end
4488   local left_stack = {}
4489   local right_stack = {}
4490   for _ , one_item in ipairs ( t ) do
4491     if one_item == "EOL" then
4492       for i = #right_stack, 1, -1 do
4493         tex.sprint ( right_stack[i] )
4494       end

```

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

```

4495     sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
4496     tex.sprint ( table.concat ( left_stack ) )
4497   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}`.

```

4498   if one_item[1] == "Open" then
4499     tex.sprint ( one_item[2] )
4500     table.insert ( left_stack , one_item[2] )
4501     table.insert ( right_stack , one_item[3] )
4502   else

```

```

4503     if one_item[1] == "Close" then
4504         tex.sprint ( right_stack[#right_stack] )
4505         left_stack[#left_stack] = nil
4506         right_stack[#right_stack] = nil
4507     else
4508         tex.tprint ( one_item )
4509     end
4510 end
4511 end
4512 end
4513 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).

```

4514 local my_file_lines
4515 function my_file_lines ( filename )
4516     local f = io.open ( filename , 'rb' )
4517     local s = f : read ( '*a' )
4518     f : close ( )

```

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

```

4519     return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
4520 end

```

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

```

4521 function piton.ReadFile ( name , first_line , last_line )
4522     local s = ''
4523     local i = 0
4524     for line in my_file_lines ( name ) do
4525         i = i + 1
4526         if i >= first_line then
4527             s = s .. '\r' .. line
4528         end
4529         if i >= last_line then break end
4530     end

```

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

```

4531     if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
4532         s = s : sub ( 5 , -1 )
4533     end
4534     sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4535     tex.sprint ( luatexbase.catcodetables.other , s )
4536     sprintL3 ( "]" )
4537 end

```

```

4538 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
4539     local s
4540     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4541     piton.GobbleParse ( lang , n , splittable , s )
4542 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 `#`.

```
4543 function piton.ParseBis ( lang , code )
4544   return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
4545 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.

```
4546 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:`.

```
4547   return piton.Parse
4548     (
4549       lang ,
4550       code : gsub ( [[\@@_breakable_space: ]] , ' ' )
4551     )
4552 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.

```
4553 local AutoGobbleLPEG =
4554   ( (
4555     P " " ^ 0 * "\r"
4556     +
4557     Ct ( C " " ^ 0 ) / table.getn
4558     * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
4559   ) ^ 0
4560   * ( Ct ( C " " ^ 0 ) / table.getn
4561       * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4562 ) / math.min
```

The following LPEG is similar but works with the tabulations.

```
4563 local TabsAutoGobbleLPEG =
4564   (
4565     (
4566       P "\t" ^ 0 * "\r"
4567       +
4568       Ct ( C "\t" ^ 0 ) / table.getn
4569       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
4570     ) ^ 0
4571     * ( Ct ( C "\t" ^ 0 ) / table.getn
4572         * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4573   ) / 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).

```

4574 local EnvGobbleLPEG =
4575     ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
4576     * 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.

```

4577 function piton.Gobble ( n , code )
4578     if n == 0 then return
4579         code
4580     else
4581         if n == -1 then
4582             n = AutoGobbleLPEG : match ( code )

```

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

```

4583         if tonumber(n) then else n = 0 end
4584     else
4585         if n == -2 then
4586             n = EnvGobbleLPEG : match ( code )
4587         else
4588             if n == -3 then
4589                 n = TabsAutoGobbleLPEG : match ( code )
4590                 if tonumber(n) then else n = 0 end
4591             end
4592         end
4593     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...

```

4594         if n == 0 then return
4595             code
4596         else return

```

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

```

4597     ( Ct (
4598         ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4599         * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4600         ) ^ 0 )
4601     / table.concat
4602     ) : match ( code )
4603 end
4604 end
4605 end

```

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

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

```

4606 function piton.GobbleParse ( lang , n , splittable , code )
4607     piton.ComputeLinesStatus ( code , splittable )
4608     piton.last_code = piton.Gobble ( n , code )
4609     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`.

```

4610     piton.CountLines ( piton.last_code )
4611     piton.Parse ( lang , piton.last_code )
4612     piton.join_and_write ( )
4613 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`.

```

4614 function piton.join_and_write ( )
4615   if piton.join ~= '' then
4616     if not piton.join_files [ piton.join ] then
4617       piton.join_files [ piton.join ] = piton.get_last_code ( )
4618     else
4619       if piton.join_separation == '' then
4620         piton.join_files [ piton.join ] =
4621           piton.join_files [ piton.join ]
4622             .. "\r\n"
4623             .. piton.get_last_code ( )
4624       else
4625         piton.join_files [ piton.join ] =
4626           piton.join_files [ piton.join ]
4627             .. "\r\n"
4628             .. ( piton.join_separation : gsub ( '##' , '#' ) )
4629             .. "\r\n"
4630             .. piton.get_last_code ( )
4631       end
4632     end
4633   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.

```

4634   if piton.write ~= '' then

```

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

```

4635     local file_name = ''
4636     if piton.path_write == '' then
4637       file_name = piton.write
4638     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.

```

4639       local attr = lfs.attributes ( piton.path_write )
4640       if attr and attr.mode == "directory" then
4641         file_name = piton.path_write .. "/" .. piton.write
4642       else

```

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

```

4643         sprintL3 [[ \@@_error_or_warning:n { InexistentDirectory } ]]
4644       end
4645     end
4646     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`).

```

4647     if not piton.write_files [ file_name ] then
4648       piton.write_files [ file_name ] = piton.get_last_code ( )
4649     else
4650       piton.write_files [ file_name ] =
4651         piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4652     end
4653   end
4654 end
4655 end

```

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

```

4656 function piton.GobbleParseNoPrint ( lang , n , code )
4657   piton.last_code = piton.Gobble ( n , code )
4658   piton.last_language = lang
4659   piton.join_and_write ( )
4660 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).

```

4661 function piton.GobbleSplitParse ( lang , n , splittable , code )
4662   local chunks
4663   chunks =
4664     (
4665       Ct (
4666         (
4667           P " " ^ 0 * "\r"
4668         +
4669           C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4670             - ( P " " ^ 0 * ( P "\r" + -1 ) )
4671           ) ^ 1
4672         )
4673       ) ^ 0
4674     )
4675   ) : match ( piton.Gobble ( n , code ) )
4676   sprintL3 [[ \begingroup ]]
4677   sprintL3
4678     (
4679       [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, }]]
4680       .. "language = " .. lang .. ", "
4681       .. "splittable = " .. splittable .. "}"
4682     )
4683   for k , v in pairs ( chunks ) do
4684     if k > 1 then
4685       sprintL3 ( [[ \l_@@_split_separation_tl ]] )
4686     end
4687     tex.print
4688       (
4689         [[\begin{}} .. piton.env_used_by_split .. "}\r"
4690         .. v
4691         .. [[\end{}} .. piton.env_used_by_split .. "}\r"
4692       )
4693   end
4694   sprintL3 [[ \endgroup ]]
4695 end

```

```

4696 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4697   local s
4698   s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4699   piton.GobbleSplitParse ( lang , n , splittable , s )
4700 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.

```

4701 piton.string_between_chunks =
4702   [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4703   .. [[ \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.

```

4704 function piton.get_last_code ( )
4705     return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4706         : gsub ( '\r\n?' , '\n' )
4707 end

```

3.14 To count the number of lines

```

4708 local CountBeamerEnvironments
4709 function CountBeamerEnvironments ( code ) return
4710     (
4711         Ct (
4712             (
4713                 P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4714                 +
4715                 ( 1 - P "\r" ) ^ 0 * "\r"
4716             ) ^ 0
4717             * ( 1 - P "\r" ) ^ 0
4718             * -1
4719         ) / table.getn
4720     ) : match ( code )
4721 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.

```

4722 function piton.CountLines ( code )
4723     local count
4724     count =
4725         ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4726             *
4727             (
4728                 space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4729                 + space ^ 0
4730             ) ^ -1
4731             * -1
4732         ) / table.getn
4733     ) : match ( code )
4734     if piton.beamer then
4735         count = count - 2 * CountBeamerEnvironments ( code )
4736     end
4737     sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4738 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).

```

4739 function piton.CountNonEmptyLines ( code )
4740     local count = 0

```

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

```

4741     count =
4742         ( Ct ( ( P " " ^ 0 * "\r"
4743             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4744             * ( 1 - P "\r" ) ^ 0
4745             * -1
4746         ) / table.getn

```

```

4747     ) : match ( code )
4748     count = count + 1
4749     if piton.beamer then
4750         count = count - 2 * CountBeamerEnvironments ( code )
4751     end
4752     sprintL3
4753     ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4754 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.

```

4755 function piton.ComputeRange ( s , t , file_name )
4756     local first_line = -1
4757     local count = 0
4758     local last_found = false
4759     for line in io.lines ( file_name ) do
4760         if first_line == -1 then
4761             if line : sub ( 1 , #s ) == s then
4762                 first_line = count
4763             end
4764         else
4765             if line : sub ( 1 , #t ) == t then
4766                 last_found = true
4767                 break
4768             end
4769         end
4770         count = count + 1
4771     end
4772     if first_line == -1 then
4773         sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4774     else
4775         if not last_found then
4776             sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4777         end
4778     end
4779     sprintL3 (
4780         [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 }'
4781         .. [[ \global \l_@@_last_line_int = ]] .. count )
4782 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`.

```

4783 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).

```

4784 local lpeg_line_beamer
4785 if piton.beamer then
4786   lpeg_line_beamer =
4787     space ^ 0
4788     * P [[\begin{}} * beamerEnvironments * "]"
4789     * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4790   +
4791     space ^ 0
4792     * P [[\end{}} * beamerEnvironments * "]"
4793 else
4794   lpeg_line_beamer = P ( false )
4795 end
4796 local lpeg_empty_lines =
4797   Ct (
4798     ( lpeg_line_beamer * "\r"
4799       +
4800       P " " ^ 0 * "\r" * Cc ( 0 )
4801       +
4802       ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4803     ) ^ 0
4804     *
4805     ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4806   )
4807   * -1
4808 local lpeg_all_lines =
4809   Ct (
4810     ( lpeg_line_beamer * "\r"
4811       +
4812       ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4813     ) ^ 0
4814     *
4815     ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4816   )
4817   * -1

```

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

```

4818 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`).

```

4819 local lines_status
4820 local s = splittable
4821 if splittable < 0 then s = - splittable end
4822 if splittable > 0 then
4823   lines_status = lpeg_all_lines : match ( code )
4824 else

```

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

```

4825   lines_status = lpeg_empty_lines : match ( code )
4826   for i , x in ipairs ( lines_status ) do
4827     if x == 0 then
4828       for j = 1 , s - 1 do
4829         if i + j > #lines_status then break end
4830         if lines_status[i+j] == 0 then break end
4831         lines_status[i+j] = 2
4832       end
4833     for j = 1 , s - 1 do

```

```

4834         if i - j == 1 then break end
4835         if lines_status[i-j-1] == 0 then break end
4836         lines_status[i-j-1] = 2
4837     end
4838 end
4839 end
4840 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.

```

4841 for j = 1 , s - 1 do
4842     if j > #lines_status then break end
4843     if lines_status[j] == 0 then break end
4844     lines_status[j] = 2
4845 end

```

Now, from the end of the code.

```

4846 for j = 1 , s - 1 do
4847     if #lines_status - j == 0 then break end
4848     if lines_status[#lines_status - j] == 0 then break end
4849     lines_status[#lines_status - j] = 2
4850 end

```

```

4851 piton.lines_status = lines_status
4852 end

4853 function piton.TranslateBeamerEnv ( code )
4854     local s
4855     s =
4856     (
4857         Ct (
4858             (
4859                 space ^ 0
4860                 * C (
4861                     ( P "\\begin{" + "\\end{" )
4862                     * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * "\r"
4863                 )
4864                 + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4865             ) ^ 0
4866             *
4867             (
4868                 (
4869                     space ^ 0
4870                     * C (
4871                         ( P "\\begin{" + "\\end{" )
4872                         * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * -1
4873                     )
4874                     + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4875                 ) ^ -1
4876             )
4877         ) ^ -1 / table.concat
4878     ) : match ( code )
4879     sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4880     tex.sprint ( luatexbase.catcodetables.other , s )
4881     sprintL3 ( "]" )
4882 end

```

3.16 To create new languages with the syntax of listings

```

4883 function piton.new_language ( lang , definition )
4884     lang = lang : lower ( )

```

```

4885 local alpha , digit = lpeg.alpha , lpeg.digit
4886 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.

```

4887 function add_to_letter ( c )
4888     if c ~= " " then table.insert ( extra_letters , c ) end
4889 end

```

For the digits, it's straitforward.

```

4890 function add_to_digit ( c )
4891     if c ~= " " then digit = digit + c end
4892 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 `}`).

```

4893 local other = S " :_@+~</>!?. ( ) ~^=#&\\"'\\$" --
4894 local extra_others = { }
4895 function add_to_other ( c )
4896     if c ~= " " then

```

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

```

4897         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 `</....>`.

```

4898         other = other + P ( c )
4899     end
4900 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`.

```

4901 local def_table
4902 if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4903     def_table = {}
4904 else
4905     local strict_braces =
4906         P { "E" ,
4907             E = ( "{" * V "F" * "}" + ( 1 - S " ,{" }" ) ) ^ 0 ,
4908             F = ( "{" * V "F" * "}" + ( 1 - S "{" }" ) ) ^ 0
4909         }
4910     local cut_definition =
4911         P { "E" ,
4912             E = Ct ( V "F" * ( " , " * V "F" ) ^ 0 ) ,
4913             F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4914                 * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4915         }
4916     def_table = cut_definition : match ( definition )
4917 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.).

```

4918 local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4919 local tex_arg = tex_braced_arg + C ( 1 )
4920 local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4921 local args_for_tag
4922     = tex_option_arg

```

```

4923     * space ^ 0
4924     * tex_arg
4925     * space ^ 0
4926     * tex_arg
4927 local args_for_morekeywords
4928   = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4929     * space ^ 0
4930     * tex_option_arg
4931     * space ^ 0
4932     * tex_arg
4933     * space ^ 0
4934     * ( tex_braced_arg + Cc ( nil ) )
4935 local args_for_moredelims
4936   = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4937     * args_for_morekeywords
4938 local args_for_morecomment
4939   = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4940     * space ^ 0
4941     * tex_option_arg
4942     * space ^ 0
4943     * 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**.

```

4944 local sensitive = true
4945 local style_tag , left_tag , right_tag
4946 for _ , x in ipairs ( def_table ) do
4947   if x[1] == "sensitive" then
4948     if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4949       sensitive = true
4950     else
4951       if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4952     end
4953   end
4954   if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4955   if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4956   if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4957   if x[1] == "tag" then
4958     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4959     style_tag = style_tag or [[\PitonStyle{Tag}]]
4960   end
4961 end

```

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

```

4962 local Number =
4963   K ( 'Number.Internal' ,
4964     ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4965       + digit ^ 0 * "." * digit ^ 1
4966       + digit ^ 1 )
4967     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4968     + digit ^ 1
4969   )
4970 local string_extra_letters = ""
4971 for _ , x in ipairs ( extra_letters ) do
4972   if not ( extra_others[x] ) then
4973     string_extra_letters = string_extra_letters .. x
4974   end
4975 end
4976 local letter = alpha + S ( string_extra_letters )
4977   + P "â" + "à" + "ç" + "é" + "ê" + "ë" + "ï" + "î"
4978   + "ô" + "û" + "ü" + "Â" + "Ã" + "Ç" + "É" + "Ê" + "Ë" + "Ï"

```

```

4979         + "ĩ" + "İ" + "Ŧ" + "Ū" + "Ū"
4980     local alphanum = letter + digit
4981     local identifier = letter * alphanum ^ 0
4982     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.

```

4983     local split_clist =
4984     P { "E" ,
4985         E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4986             * ( P "{" ) ^ 1
4987             * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4988             * ( P "}" ) ^ 1 * space ^ 0 ,
4989         F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4990     }

```

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

```

4991     local keyword_to_lpeg
4992     function keyword_to_lpeg ( name ) return
4993         Q ( Cmt (
4994             C ( identifier ) ,
4995             function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4996             end
4997         )
4998     )
4999     end
5000     local Keyword = P ( false )
5001     local PrefixedKeyword = P ( false )

```

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

```

5002     local keywords_prefix = ''

```

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

```

5003     for _ , x in ipairs ( def_table )
5004     do if x[1] == "morekeywords"
5005         or x[1] == "otherkeywords"
5006         or x[1] == "moredirectives"
5007         or x[1] == "moretexcs"
5008     then
5009         local keywords = P ( false )
5010         local style = [[\PitonStyle{Keyword}]]
5011         if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
5012         style = tex_option_arg : match ( x[2] ) or style
5013         local n = tonumber ( style )
5014         if n then
5015             if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
5016             end
5017         for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
5018             if x[1] == "moretexcs" then
5019                 keywords = Q ( [[\]] .. word ) + keywords
5020             else
5021                 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*.

```

5022                 then keywords = Q ( word ) + keywords
5023                 else keywords = keyword_to_lpeg ( word ) + keywords
5024             end
5025         end
5026     end
5027     Keyword = Keyword +
5028     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"

```

5029 end

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

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

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

```
5030     if x[1] == "keywordsprefix" then
5031         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
5032         PrefixedKeyword = PrefixedKeyword
5033             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
5034         keywords_prefix = keywords_prefix .. prefix
5035     end
5036 end
```

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

```
5037     local long_string = P ( false )
5038     local Long_string = P ( false )
5039     local LongString = P ( false )
5040     local central_pattern = P ( false )
5041     for _ , x in ipairs ( def_table ) do
5042         if x[1] == "morestring" then
5043             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
5044             arg2 = arg2 or [[\PitonStyle{String.Long}]]
5045             if arg1 ~= "s" then
5046                 arg4 = arg3
5047             end
5048             central_pattern = 1 - S ( " \r" .. arg4 )
5049             if arg1 : match "b" then
5050                 central_pattern = P ( [[\]] .. arg3 ) + central_pattern
5051             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...

```
5052         if arg1 : match "d" or arg1 == "m" then
5053             central_pattern = P ( arg3 .. arg3 ) + central_pattern
5054         end
5055         if arg1 == "m"
5056         then prefix = B ( 1 - letter - ")" - "]" )
5057         else prefix = P ( true )
5058         end
```

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

```
5059     long_string = long_string +
5060         prefix
5061         * arg3
5062         * ( space + central_pattern ) ^ 0
5063         * arg4
```

Now a pattern *with captures*.

```
5064     local pattern =
5065         prefix
5066         * Q ( arg3 )
5067         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
5068         * Q ( arg4 )
```

We will need `Long_string` in the nested comments.

```
5069     Long_string = Long_string + pattern
5070     LongString = LongString +
5071         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
```

```

5072         * pattern
5073         * Ct ( Cc "Close" )
5074     end
5075 end

```

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

```

5076 local braces = Compute_braces ( long_string )
5077 if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
5078
5079 DetectedCommands =
5080     Compute_DetectedCommands ( lang , braces )
5081     + Compute_RawDetectedCommands ( lang , braces )
5082
5083 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*).

```

5084 local sComment = P ( false )

```

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

```

5085 local lComment = P ( false )

```

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

```

5086 local fComment = P ( false )
5087 local fComment = P ( false )
5088
5089 for _ , x in ipairs ( def_table ) do
5090     if x[1] == "morecomment" then
5091         local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
5092         arg2 = arg2 or [[\PitonStyle{Comment}]]

```

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

```

5093     if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
5094     if arg1 : match "l" or arg1 : match "f" then
5095         local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
5096                     : match ( other_args )
5097         if arg3 == [[\#]] then arg3 = "#" end -- mandatory
5098         if arg3 == [[\%]] then arg3 = "%" end -- mandatory
5099         local MyLPEG =
5100             Ct ( Cc "Open"
5101                 * Cc ( "{" .. arg2 .. [[{\PitonSpaceSubstitute{}}] ]
5102                     * Cc "}" )
5103             )
5104             * Q ( arg3 )
5105             * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $ noqa
5106             * Ct ( Cc "Close" )
5107             * ( EOL + -1 )
5108         if arg1 : match "l" then
5109             lComment = lComment + MyLPEG
5110         else
5111             fComment = fComment + MyLPEG
5112         end
5113     else
5114         local arg3 , arg4 =
5115             ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
5116         if arg1 : match "s" then
5117             sComment = sComment +
5118                 Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
5119                 * Q ( arg3 )
5120                 * (

```

```

5121         CommentMath
5122         + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
5123         + EOL
5124     ) ^ 0
5125     * Q ( arg4 )
5126     * Ct ( Cc "Close" )
5127 end
5128 if arg1 : match "n" then
5129     sComment = sComment +
5130     Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
5131     * P { "A" ,
5132         A = Q ( arg3 )
5133         * ( V "A"
5134             + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
5135                 - S "\r$" ) ^ 1 ) -- $
5136             + long_string
5137             + "$" -- $
5138             * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
5139             * "$" -- $
5140             + EOL
5141         ) ^ 0
5142         * Q ( arg4 )
5143     }
5144     * Ct ( Cc "Close" )
5145 end
5146 end
5147 end

```

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

```

5148 if x[1] == "moredelim" then
5149     local arg1 , arg2 , arg3 , arg4 , arg5
5150     = args_for_moredelims : match ( x[2] )
5151     local MyFun = Q
5152     if arg1 == "*" or arg1 == "**" then
5153         function MyFun ( x )
5154             if x ~= '' then return
5155                 LPEG1[lang] : match ( x )
5156             end
5157         end
5158     end
5159     local left_delim
5160     if arg2 : match "i" then
5161         left_delim = P ( arg4 )
5162     else
5163         left_delim = Q ( arg4 )
5164     end
5165     if arg2 : match "l" then
5166         sComment = sComment +
5167         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" ) * Cc "}" )
5168         * left_delim
5169         * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
5170         * Ct ( Cc "Close" )
5171         * ( EOL + -1 )
5172     end
5173     if arg2 : match "s" then
5174         local right_delim
5175         if arg2 : match "i" then
5176             right_delim = P ( arg5 )
5177         else
5178             right_delim = Q ( arg5 )
5179         end
5180         sComment = sComment +
5181         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" ) * Cc "}" )
5182         * left_delim

```

```

5183         * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
5184         * right_delim
5185         * Ct ( Cc "Close" )
5186     end
5187 end
5188 end
5189
5190 local Delim = Q ( S "{[()]}" )
5191 local Punct = Q ( S "=",:;!\\'\"" )
5192
5193 local EOL =
5194     P "\r"
5195     *
5196     (
5197         space ^ 0 * -1
5198         +
5199         Cc "EOL"
5200     )
5201     * ( ( fComment + lComment ) ^ 0 * LeadingSpace ^ 0 * # ( 1 - S " \r" ) ) ^ -1

```

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

```

5201 local lpeg_central = 1 - S " '\r[{}]" - digit - S ( keywords_prefix )
5202 if piton.begin_escape then
5203     lpeg_central = lpeg_central - piton.begin_escape
5204 end
5205 if piton.begin_escape_math then
5206     lpeg_central = lpeg_central - piton.begin_escape_math
5207 end
5208 local Word = Q ( lpeg_central ^ 1 )
5209
5210 local Main =
5211     space ^ 0 * EOL * ( fComment + lComment ) ^ 0
5212     + Space
5213     + Tab
5214     + Escape + EscapeMath
5215     + CommentLaTeX
5216     + Beamer
5217     + DetectedCommands
5218     + sComment
5219     + lComment

```

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

```

5219     + LongString
5220     + Delim
5221     + PrefixedKeyword
5222     + Keyword * ( -1 + # ( 1 - alphanum ) )
5223     + Punct
5224     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5225     + Number
5226     + 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.

```

5227 LPEG1[lang] = Main ^ 0

```

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

```

5228 LPEG2[lang] =
5229     Ct (
5230         ( space ^ 0 * P "\r" ) ^ -1
5231         * Lc [[ \@@_begin_line: ]]

```

```

5232      * ( fComment + lComment ) ^ 0 -- 2026-05-12
5233      * LeadingSpace ^ 0
5234      * ( space ^ 1 * -1 + Main ) ^ 0
5235      * -1
5236      * Lc [[ \@@_end_line: ]]
5237    )

```

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

```

5238    if left_tag then
5239      local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
5240      * Q ( left_tag * other ^ 0 ) -- $
5241      * ( ( 1 - P ( right_tag ) ) ^ 0 )
5242      / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
5243      * Q ( right_tag )
5244      * Ct ( Cc "Close" )
5245    MainWithoutTag
5246      = space ^ 1 * -1
5247      + space ^ 0 * EOL
5248      + Space
5249      + Tab
5250      + Escape + EscapeMath
5251      + CommentLaTeX
5252      + Beamer
5253      + DetectedCommands
5254      + sComment
5255      + Delim
5256      + LongString
5257      + PrefixedKeyword
5258      + Keyword * ( -1 + # ( 1 - alphanum ) )
5259      + Punct
5260      + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
5261      + Number
5262      + Word
5263    LPEG0[lang] = MainWithoutTag ^ 0
5264    local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
5265      + Beamer + DetectedCommands + sComment + Tag
5266    MainWithTag
5267      = space ^ 1 * -1
5268      + space ^ 0 * EOL
5269      + Space
5270      + LPEGaux
5271      + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
5272    LPEG1[lang] = MainWithTag ^ 0
5273    LPEG2[lang] =
5274      Ct (
5275        ( space ^ 0 * P "\r" ) ^ -1
5276        * Lc [[ \@@_begin_line: ]]
5277        * Beamer
5278        * LeadingSpace ^ 0
5279        * LPEG1[lang]
5280        * -1
5281        * Lc [[ \@@_end_line: ]]
5282      )
5283  end
5284 end

```

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

```

5285 function piton.write_files_now ( )
5286   for file_name , file_content in pairs ( piton.write_files ) do
5287     local file = io.open ( file_name , "w" )

```

```

5288     if file then
5289         file : write ( file_content )
5290         file : close ( )
5291     else
5292         sprintL3
5293         ( [[ \@@_error_or_warning:nn { FileError } { ]] .. file_name .. "}" )
5294     end
5295 end
5296 end

```

3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

5297 function piton.utf16 ( str )
5298     local hex = { "FEFF" } -- BOM UTF-16BE
5299     for _, codepoint in utf8.codes(str) do
5300         table.insert(hex, string.format("%04X", codepoint))
5301     end
5302     return table.concat(hex)
5303 end
5304 </LUA>

```